

3. vector 迭代器的自增和解引用运算

迭代器类型定义了一些操作来获取迭代器所指向的元素，并允许程序员将迭代器从一个元素移动到另一个元素。

迭代器类型可使用解引用操作符（*操作符）来访问迭代器所指向的元素：

```
*iter = 0;
```

解引用操作符返回迭代器当前所指向的元素。假设 `iter` 指向 `vector` 对象 `ivec` 的第一个元素，那么 `*iter` 和 `ivec[0]` 就是指向同一个元素。上面这个语句的效果就是把这个元素的值赋为 0。

迭代器使用自增操作符（1.4.1 节）向前移动迭代器指向容器中下一个元素。从逻辑上说，迭代器的自增操作和 `int` 型对象的自增操作类似。对 `int` 对象来说，操作结果就是把 `int` 型值“加 1”，而对迭代器对象则是把容器中的迭代器“向前移动一个位置”。因此，如果 `iter` 指向第一个元素，则 `++iter` 指向第二个元素。



由于 `end` 操作返回的迭代器不指向任何元素，因此不能对它进行解引用或自增操作。

4. 迭代器的其他操作

另一对可执行于迭代器的操作就是比较：用 `==` 或 `!=` 操作符来比较两个迭代器，如果两个迭代器对象指向同一个元素，则它们相等，否则就不相等。

5. 迭代器应用的程序示例

假设已声明了一个 `vector<int>` 型的 `ivec` 变量，要把它所有元素值重置为 0，可以用下标操作来完成：

```
// reset all the elements in ivec to 0
for (vector<int>::size_type ix = 0; ix != ivec.size(); ++ix)
    ivec[ix] = 0;
```

上述程序用 `for` 循环遍历 `ivec` 的元素，`for` 循环定义了一个索引 `ix`，每循环迭代一次 `ix` 就自增 1。`for` 循环体将 `ivec` 的每个元素赋值为 0。

更典型的做法是用迭代器来编写循环：

```
// equivalent loop using iterators to reset all the elements in ivec to 0
for (vector<int>::iterator iter = ivec.begin();
     iter != ivec.end(); ++iter)
    *iter = 0; // set element to which iter refers to 0
```

`for` 循环首先定义了 `iter`，并将它初始化为指向 `ivec` 的第一个元素。`for` 循环的条件测试 `iter` 是否与 `end` 操作返回的迭代器不等。每次迭代 `iter` 都自增 1，这个 `for` 循环的效果是从 `ivec` 第一个元素开始，顺序处理 `vector` 中的每一元素。最后，`iter` 将指向 `ivec` 中的最后一个元素，处理完最后一个元素后，`iter` 再增加 1，就会与 `end` 操作的返回值相等，在这种情况下，循环终止。

`for` 循环体内的语句用解引用操作符来访问当前元素的值。和下标操作符一样，解引用操作符的返回值是一个左值，因此可以对它进行赋值来改变它的值。上述循环的效果就是把 `ivec` 中所有元素都赋值为 0。

通过上述对代码的详细分析，可以看出这段程序与用下标操作符的版本达到相同的操作效果：从 vector 的第一个元素开始，把 vector 中每个元素都置为 0。



本节给出的例子程序和 3.3.2 节 vector 的下标操作的程序一样，如果 vector 为空，程序是安全的。如果 ivec 为空，则 begin 返回的迭代器不指向任何元素——由于没有元素，所以它不能指向任何元素。在这种情况下，从 begin 操作返回的迭代器与从 end 操作返回的迭代器的值相同，因此 for 语句中的测试条件立即失败。

6. const_iterator

前面的程序用 vector::iterator 改变 vector 中的元素值。每种容器类型还定义了一种名为 const_iterator 的类型，该类型只能用于读取容器内元素，但不能改变其值。

当我们对普通 iterator 类型解引用时，得到对某个元素的非 const 引用（2.5 节）。而如果我们使用 const_iterator 类型解引用时，则可以得到一个指向 const 对象的引用（2.4 节），如同任何常量一样，该对象不能进行重写。

例如，如果 text 是 vector<string> 类型，程序员想要遍历它，输出每个元素，可以这样编写程序：

```
// use const_iterator because we won't change the elements
for (vector<string>::const_iterator iter = text.begin();
     iter != text.end(); ++iter)
    cout << *iter << endl; // print each element in text
```

除了是从迭代器读取元素值而不是对它进行赋值之外，这个循环与前一个相似。由于这里只需要借助迭代器进行读，不需要写，这里把 iter 定义为 const_iterator 类型。当对 const_iterator 类型解引用时，返回的是一个 const 值。不允许用 const_iterator 进行赋值：

```
for (vector<string>::const_iterator iter = text.begin();
     iter != text.end(); ++ iter)
    *iter = " "; // error: *iter is const
```

使用 const_iterator 类型时，我们可以得到一个迭代器，它自身的值可以改变，但不能用来改变其所指向的元素的值。可以对迭代器进行自增以及使用解引用操作符来读取值，但不能对该元素值赋值。

不要把 const_iterator 对象与 const 的 iterator 对象混淆起来。声明一个 const 迭代器时，必须初始化迭代器。一旦被初始化后，就不能改变它的值：

```
vector<int> nums(10); // nums is nonconst
const vector<int>::iterator cit = nums.begin();
*cit = 1; // ok: cit can change its underlying element
++cit; // error: can't change the value of cit
```

const_iterator 对象可以用于 const vector 或非 const vector，因为不能改写元素值。const 迭代器这种类型几乎没什么用处：一旦它被初始化后，只能用它来改写其指向的元素，但不能使它指向任何其他元素：

```
const vector<int> nines(10, 9); // cannot change elements in nines
// error: cit2 could change the element it refers to and nines is const
```

```
const vector<int>::iterator cit2 = nines.begin();
// ok: it can't change an element value, so it can be used with a const vector<int>
vector<int>::const_iterator it = nines.begin();
*it = 10; // error: *it is const
++it;     // ok: it isn't const so we can change its value
```



```
// an iterator that cannot write elements
vector<int>::const_iterator
// an iterator whose value cannot change
const vector<int>::iterator
```

习题

- 习题 3.17 重做 3.3.2 节的习题，用迭代器而不是下标操作来访问 `vector` 中的元素。
- 习题 3.18 编写程序来创建有 10 个元素的 `vector` 对象。用迭代器把每个元素值改为当前值的 2 倍。
- 习题 3.19 验证习题 3.18 的程序，输出 `vector` 的所有元素。
- 习题 3.20 解释一下在上几个习题的程序实现中你用了哪种迭代器，并说明原因。
- 习题 3.21 何时使用 `const` 迭代器的？又在何时使用 `const_iterator`？解释两者的区别。

迭代器的算术操作

除了一次移动迭代器的一个元素的增量操作符外，`vector` 迭代器（其他标准库容器迭代器很少）也支持其他的算术操作。这些操作称为**迭代器算术操作**（`iterator arithmetic`），包括：

100

- `iter + n`

`iter - n`

可以对迭代器对象加上或减去一个整型值。这样做将产生一个新的迭代器，其位置在 `iter` 所指元素之前（加）或之后（减）`n` 个元素的位置。加或减之后的结果必须指向 `iter` 所指 `vector` 中的某个元素，或者是 `vector` 末端的后一个元素。加上或减去的值的类型应该是 `vector` 的 `size_type` 或 `difference_type` 类型（参考下面的解释）。

- `iter1 - iter2`

该表达式用来计算两个迭代器对象的距离，该距离是名为 `difference_type` 的 `signed` 类型的值，这里的 `difference_type` 类型类似于 `size_type` 类型，也是由 `vector` 定义的。`difference_type` 是 `signed` 类型，因为减法运算可能产生负数的结果。该类型可以保证足够大以存储任何两个迭代器对象间的距离。`iter1` 与 `iter2` 两者必须都指向同一 `vector` 中的元素，或者指向 `vector` 末端之后的下一个元素。

可以用迭代器算术操作来移动迭代器直接指向某个元素，例如，下面语句直接定位于 `vector` 的中间元素：

```
vector<int>::iterator mid = vi.begin() + vi.size()/2;
```

上述代码用来初始化 `mid`，使其指向 `vi` 中最靠近正中间的元素。这种直接计算迭代器的方法，与用迭代器逐个元素自增操作到达中间元素的方法是等价的，但前者的效率要高得多。



任何改变 `vector` 长度的操作都会使已存在的迭代器失效。例如，在调用 `push_back` 之后，就不能再信赖指向 `vector` 的迭代器的值了。

习题

习题 3.22 如果采用下面的方法来计算 `mid` 会产生什么结果？

```
vector<int>::iterator mid = (vi.begin() + vi.end())/2;
```

3.5 标准库 `bitset` 类型

有些程序要处理二进制位的有序集，每个位可能包含 0（关）值或 1（开）值。位是用来保存一组项或条件的 yes/no 信息（有时也称标志）的简洁方法。标准库提供的 `bitset` 类简化了位集的处理。要使用 `bitset` 类就必须包含相关的头文件。在本书提供的例子中，假设都已使用 `std::bitset` 的 `using` 声明：

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

101

3.5.1 `bitset` 对象的定义和初始化

表 3-6 列出了 `bitset` 的构造函数。类似于 `vector`，`bitset` 类是一种类模板；而与 `vector` 不一样的是 `bitset` 类型对象的区别仅在其长度而不在其类型。在定义 `bitset` 时，要明确 `bitset` 含有多少位，须在尖括号内给出它的长度值：

```
bitset<32> bitvec; // 32 bits, all zero
```

给出的长度值必须是常量表达式（2.7 节）。正如这里给出的，长度值必须定义为整型字面值常量或是已用常量值初始化的整型的 `const` 对象。

这条语句把 `bitvec` 定义为含有 32 个位的 `bitset` 对象。和 `vector` 的元素一样，`bitset` 中的位是没有命名的，程序员只能按位置来访问它们。位集合的位置编号从 0 开始，因此，`bitvec` 的位序是从 0 到 31。以 0 位开始的位串是低阶位（low-order bit），以 31 位结束的位串是高阶位（high-order bit）。

表 3-6 初始化 `bitset` 对象的方法

<code>bitset<n> b;</code>	<code>b</code> 有 <code>n</code> 位，每位都为 0
<code>bitset<n> b(u);</code>	<code>b</code> 是 <code>unsigned long</code> 型 <code>u</code> 的一个副本
<code>bitset<n> b(s);</code>	<code>b</code> 是 <code>string</code> 对象 <code>s</code> 中含有的位串的副本
<code>bitset<n> b(s, pos, n);</code>	<code>b</code> 是 <code>s</code> 中从位置 <code>pos</code> 开始的 <code>n</code> 个位的副本

1. 用 unsigned 值初始化 bitset 对象

当用 unsigned long 值作为 bitset 对象的初始值时, 该值将转化为二进制的位模式。而 bitset 对象中的位集作为这种位模式的副本。如果 bitset 类型长度大于 unsigned long 值的二进制位数, 则其余的高阶位将置为 0; 如果 bitset 类型长度小于 unsigned long 值的二进制位数, 则只使用 unsigned 值中的低阶位, 超过 bitset 类型长度的高阶位将被丢弃。

在 32 位 unsigned long 的机器上, 十六进制值 0xffff 表示为二进制位就是十六个 1 和十六个 0 (每个 0xf 可表示为 1111)。可以用 0xffff 初始化 bitset 对象:

```
// bitvec1 is smaller than the initializer
bitset<16> bitvec1(0xffff);           // bits 0 ... 15 are set to 1
// bitvec2 same size as initializer
bitset<32> bitvec2(0xffff);           // bits 0 ... 15 are set to 1; 16 ... 31 are 0
// on a 32-bit machine, bits 0 to 31 initialized from 0xffff
bitset<128> bitvec3(0xffff);          // bits 32 through 127 initialized to zero
```

上面的三个例子中, 0 到 15 位都置为 1。由于 bitvec1 位数少于 unsigned long 的位数, 因此 bitvec1 的初始值的高阶位被丢弃。bitvec2 和 unsigned long 长度相同, 因此所有位正好放置了初始值。bitvec3 长度大于 32, 31 位以上的高阶位就被置为 0。

2. 用 string 对象初始化 bitset 对象

当用 string 对象初始化 bitset 对象时, string 对象直接表示为位模式。从 string 对象读入位集的顺序是从右向左:

```
string strval("1100");
bitset<32> bitvec4(strval);
```

bitvec4 的位模式中第 2 和 3 的位置为 1, 其余位置都为 0。如果 string 对象的字符个数小于 bitset 类型的长度, 则高阶位将置为 0。

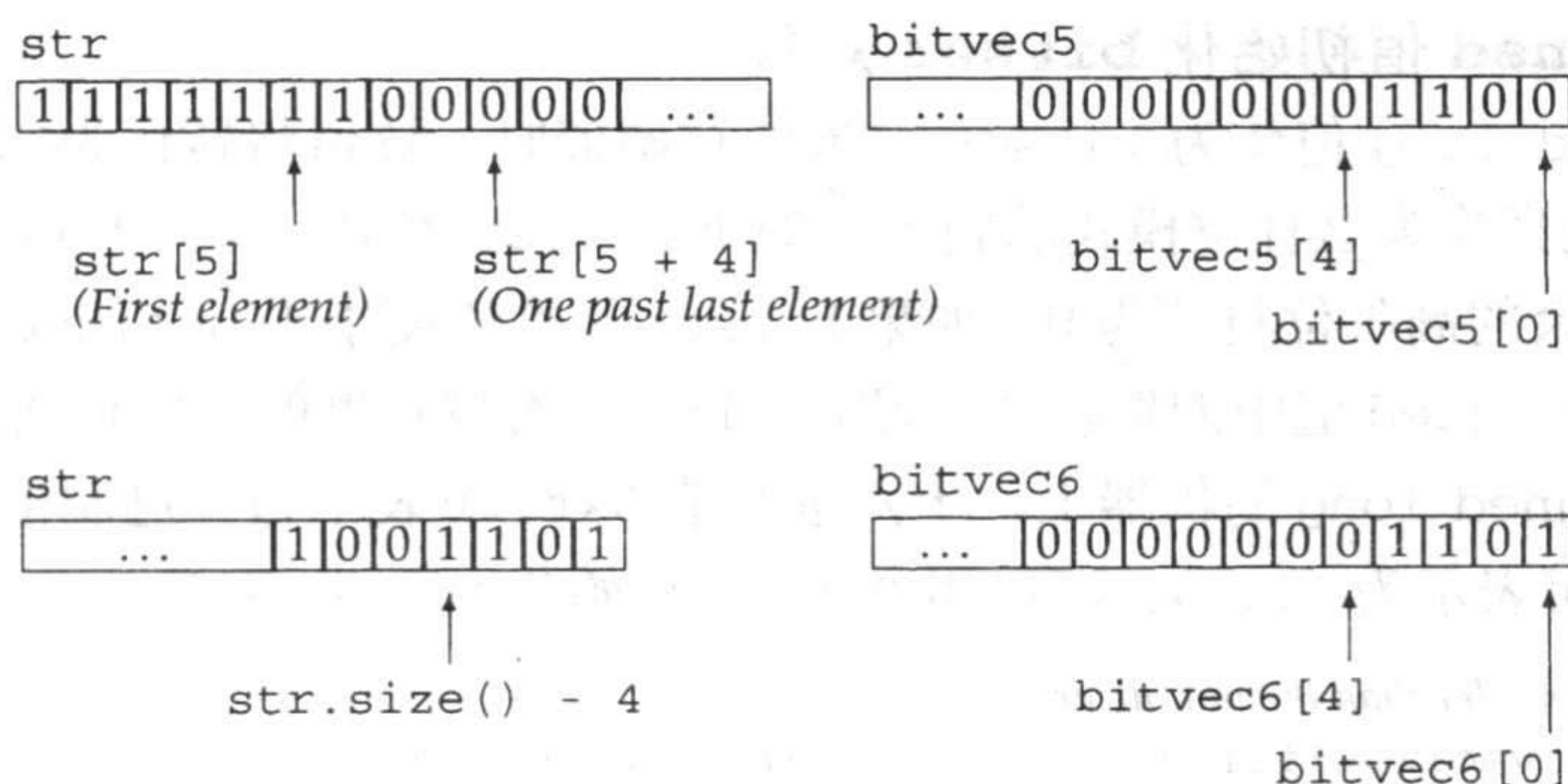


string 对象和 bitset 对象之间是反向转化的: string 对象的最右边字符 (即下标最大的那个字符) 用来初始化 bitset 对象的低阶位 (即下标为 0 的位)。当用 string 对象初始化 bitset 对象时, 记住这一差别很重要。

不一定要把整个 string 对象都作为 bitset 对象的初始值。相反, 可以只用某个子串作为初始值:

```
string str("1111111000000011001101");
bitset<32> bitvec5(str, 5, 4); // 4 bits starting at str[5], 1100
bitset<32> bitvec6(str, str.size() - 4); // use last 4 characters
```

这里用 str 中从 str[5] 开始包含四个字符的子串来初始化 bitvec5。照常, 初始化 bitset 对象时总是从子串最右边结尾字符开始的, bitvec5 的从 3 到 0 的二进制位置为 1100, 其他二进制位都置为 0。如果省略第三个参数则意味着取从开始位置一直到 string 末尾的所有字符。本例中, 取出 str 末尾的四位来对 bitvec6 的低四位进行初始化。bitvec6 其余的位初始化为 0。这些初始化过程的图示如下:



103

3.5.2 bitset 对象上的操作

多种 bitset 操作（表 3-7）用来测试或设置 bitset 对象中的单个或多个二进制位。

表 3-7 bitset 操作

b.any()	b 中是否存在置为 1 的二进制位？
b.none()	b 中不存在置为 1 的二进制位吗？
b.count()	b 中置为 1 的二进制位的个数
b.size()	b 中二进制位的个数
b[pos]	访问 b 中在 pos 处的二进制位
b.test(pos)	b 中在 pos 处的二进制位是否为 1？
b.set()	把 b 中所有二进制位都置为 1
b.set(pos)	把 b 中在 pos 处的二进制位置为 1
b.reset()	把 b 中所有二进制位都置为 0
b.reset(pos)	把 b 中在 pos 处的二进制位置为 0
b.flip()	把 b 中所有二进制位逐位取反
b.flip(pos)	把 b 中在 pos 处的二进制位取反
b.to_ulong()	用 b 中同样的二进制位返回一个 unsigned long 值
os << b	把 b 中的位集输出到 os 流

1. 测试整个 bitset 对象

如果 bitset 对象中有一个或几个二进制位置为 1，则 any 操作返回 true，也就是说，其返回值等于 1；相反，如果 bitset 对象中的二进制位全为 0，则 none 操作返回 true。

```
bitset<32> bitvec; // 32 bits, all zero
bool is_set = bitvec.any(); // false, all bits are zero
bool is_not_set = bitvec.none(); // true, all bits are zero
```

如果需要知道置为 1 的二进制位的个数，可以使用 count 操作，该操作返回置为 1 的二进制位的个数：

```
size_t bits_set = bitvec.count(); // returns number of bits that are on
```

count 操作的返回类型是标准库中命名为 **size_t** 的类型。size_t 类型定义在 cstdint 头文件

中，该文件是 C 标准库的头文件 `stddef.h` 的 C++ 版本。它是一个与机器相关的 *unsigned* 类型，其大小足以保证存储内存中对象的大小。

与 `vector` 和 `string` 中的 `size` 操作一样，`bitset` 的 `size` 操作返回 `bitset` 对象中二进制位的个数，返回值的类型是 `size_t`：

```
size_t sz = bitvec.size(); // returns 32
```

104

2. 访问 `bitset` 对象中的位

可以用下标操作符来读或写某个索引位置的二进制位，同样地，也可以用下标操作符测试给定二进制位的值或设置某个二进制位的值：

```
// assign 1 to even numbered bits
for (int index = 0; index != 32; index += 2)
    bitvec[index] = 1;
```

上面的循环把 `bitvec` 中的偶数下标的位都置为 1。

除了用下标操作符，还可以用 `set`、`test` 和 `reset` 操作来测试或设置给定二进制位的值：

```
// equivalent loop using set operation
for (int index = 0; index != 32; index += 2)
    bitvec.set(index);
```

为了测试某个二进制位是否为 1，可以用 `test` 操作或者测试下标操作符的返回值：

```
if (bitvec.test(i))
    // bitvec[i] is on
// equivalent test using subscript
if (bitvec[i])
    // bitvec[i] is on
```

如果下标操作符测试的二进制位为 1，则返回的测试值的结果为 `true`，否则返回 `false`。

3. 对整个 `bitset` 对象进行设置

`set` 和 `reset` 操作分别用来对整个 `bitset` 对象的所有二进制位全置 1 和全置 0：

```
bitvec.reset(); // set all the bits to 0.
bitvec.set();   // set all the bits to 1
```

`flip` 操作可以对 `bitset` 对象的所有位或个别位取反：

```
bitvec.flip(0); // reverses value of first bit
bitvec[0].flip(); // also reverses the first bit
bitvec.flip(); // reverses value of all bits
```

4. 获取 `bitset` 对象的值

`to_ulong` 操作返回一个 `unsigned long` 值，该值与 `bitset` 对象的位模式存储值相同。仅当 `bitset` 类型的长度小于或等于 `unsigned long` 的长度时，才可以使用 `to_ulong` 操作：

```
unsigned long ulong = bitvec3.to_ulong();
cout << "ulong = " << ulong << endl;
```

`to_ulong` 操作主要用于把 `bitset` 对象转到 C 风格或标准 C++ 之前风格的程序上。如果 `bitset` 对象包含的二进制位数超过 `unsigned long` 的长度，将会产生运行时异常。本书将在 6.13 节介绍异常 (exception)，并在 17.1 节中详细地讨论它。

105

5. 输出二进制位

可以用输出操作符输出 `bitset` 对象中的位模式：

```
bitset<32> bitvec2(0xffff); // bits 0... 15 are set to 1; 16... 31 are 0
cout << "bitvec2: " << bitvec2 << endl;
```

输出结果为：

```
bitvec2: 00000000000000000111111111111111
```

6. 使用位操作符

`bitset` 类也支持内置的位操作符。C++定义的这些操作符都只适用于整型操作数，它们所提供的操作类似于本节所介绍的 `bitset` 操作。5.3 节将介绍这些操作符。

习题

习题 3.23 解释下面每个 `bitset` 对象包含的位模式：

- (a) `bitset<64> bitvec(32);`
- (b) `bitset<32> bv(1010101);`
- (c) `string bstr; cin >> bstr; bitset<8> bv(bstr);`

习题 3.24 考虑这样的序列 1, 2, 3, 5, 8, 13, 21, 并初始化一个将该序列数字所对应的位置置为 1 的 `bitset<32>` 对象。然后换个方法，给定一个空的 `bitset` 对象，编写一小段程序把相应的数位设置为 1。

106

小结

C++标准库定义了几种更高级的抽象数据类型，包括 `string` 和 `vector` 类型。`string` 类型提供了变长的字符串，而 `vector` 类型则可用于管理同一类型的对象集合。

迭代器实现了对存储于容器中对象的间接访问。迭代器可以用于访问和遍历 `string` 类型和 `vector` 类型的元素。

下一章将介绍 C++的内置数据类型：数组和指针。这两种类型提供了类似于 `vector` 和 `string` 标准库类型的低级抽象类型。总的来说，相对于 C++内置数据类型的数组和指针而言，程序员应优先使用标准库类类型。

术语

abstract data type (抽象数据类型) 隐藏其实现的数据类型。使用抽象数据类型时，只需要了解该类型所支持的操作。

bitset 一种标准库类型，用于保存位集，并提供对各个位的测试和置位操作。

cctype header (cctype 头文件) 从 C 标准库继承而来的头文件，包含一组测试字符值的例程。8.3.4 节的表 3-3 列出了常用的例程。

class template (类模板) 一个可创建许多潜在类类型的蓝图。使用类模板时，必须给出实

际的类型和值。例如，`vector` 类型是保存给定类型对象的模板。创建一个 `vector` 对象时，必须指出这个 `vector` 对象所保存的元素的类型。`vector<int>` 保存 `int` 的对象，而 `vector<string>` 则保存 `string` 对象，以此类推。

container (容器) 一种类型，其对象保存一组给定类型的对象的集合。

difference_type 一种由 `vector` 类型定义的 `signed` 整型，用于存储任意两个迭代器间的距离。

empty 由 `string` 类型和 `vector` 类型定义的成员函数。`empty` 返回布尔值，用于检测 `string` 是否有字符或 `vector` 是否有元素。如果 `string` 或 `vector` 的 `size` 为 0，则返回 `true`，否则返回 `false`。

getline `string` 头文件中定义的函数，该函数接受一个 `istream` 对象和一个 `string` 对象，读取输入流直到下一个换行符，存储读入的输入流到 `string` 对象中，并返回 `istream` 对象。换行符被读入并丢弃。

high-order (高阶) `bitset` 对象中索引值最大的位。

index (索引) 下标操作符所使用的值，用于表示从 `string` 对象或 `vector` 对象中获取的元素。也称“下标”。

iterator (迭代器) 用于对容器类型的元素进行检查和遍历的数据类型。

iterator arithmetic (迭代器的算术操作) 应用于一些（并非全部）迭代器类型的算术操作。迭代器对象可以加上或减去一个整型数值，结果迭代器指向处于原迭代器之前或之后若干个元素的位置。两个迭代器对象可以相减，得到的结果是它们之间的距离。迭代器算术操作只适用于指向同一容器中的元素或指向容器

末端的下一元素迭代器。

low-order (低阶) `bitset` 对象中索引值最小的位。

off-the-end iterator (超出末端的迭代器) 由 `end` 操作返回的迭代器，是一种指向容器末端之后的不存在元素的迭代器。

push_back 由 `vector` 类型定义的成员函数，用于把元素追加到 `vector` 对象的尾部。

sentinel (哨兵) 一种程序设计技术，使用一个值来控制处理过程。在本章中使用由 `end` 操作返回的迭代器作为保护符，当处理完 `vector` 对象中的所有元素后，用它来停止处理 `vector` 中的元素。

size 由库类型 `string`、`vector` 和 `bitset` 定义的函数，分别用于返回此三个类型的字符个数、元素个数、二进制位的个数。`string` 和 `vector` 类的 `size` 成员函数返回 `size_type` 类型的值（例如，`string` 对象的 `size` 操作返回 `string::size_type` 类型值）。`bitset` 对象的 `size` 操作返回 `size_t` 类型值。

size_t 在 `cstdint` 头文件中定义的机器相关的无符号整型，该类型足以保存最大数组的长度。

size_type 由 `string` 类类型和 `vector` 类类型定义的类型，用以保存任意 `string` 对象或 `vector` 对象的长度。标准库类型将 `size_type` 定义为 `unsigned` 类型。

using declaration (using 声明) 使命名空间的名字可以直接引用。比如：

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

可以直接访问 `name` 而无须前缀 `namespace::`。

value initialization (值初始化) 当给定容器的长度，但没有显式提供元素的初始式时，

对容器元素进行的初始化。元素被初始化为一个编译器产生的值的副本。如果容器保存内置类型变量，则元素的初始值将置为 0。如果容器用于保存类对象，则元素的初始值由类的默认构造函数产生。只有类提供了默认构造函数时，类类型的容器元素才能进行值初始化。

++ operator (++)操作符 迭代器类型定义的自增操作符，通过“加 1”移动迭代器指向下一个元素。

:: operator (::操作符) 作用域操作符。:: 操作符在其左操作数的作用域内找到其右操作数的名字。用于访问某个命名空间中的名字，如 `std::cout`，表明名字 `cout` 来自命名空间 `std`。同样地，可用来从某个类取名字，如 `string::size_type`，表明 `size_type` 是由 `string` 类定义的。

[] operator ([])操作符 由 `string`、`vector` 和 `bitset` 类型定义的重载操作符。它接受两个操作数：左操作数是对象名字，右操作数是一个索引。该操作符用于取出位置与索引相符

的元素，索引计数从 0 开始，即第一个元素的索引为 0，最后一个元素的索引为 `obj.size() - 1`。下标操作返回左值，因此可将下标操作作为赋值操作的左操作数。对下标操作的结果赋值是赋一个新值到相应的元素。

*** operator (*操作符)** 迭代器类型定义了解引用操作符来返回迭代器所指向的对象。解引用返回左值，因此可将解引用操作符用作赋值操作的左操作数。对解引用操作的结果赋值是赋一个新值到相应的元素。

<< operator (<<操作符) 标准库类型 `string` 和 `bitset` 定义了输出操作符。`string` 类型的输出操作符将输出 `string` 对象中的字符。`bitset` 类型的输出操作符则输出 `bitset` 对象的位模式。

>> operator (>>操作符) 标准库类型 `string` 和 `bitset` 定义了输入操作符。`string` 类型的输入操作符读入以空白字符为分隔符的字符串，并把读入的内容存储在右操作数 (`string` 对象) 中。`bitset` 类型的输入操作符则读入一个位序列到其 `bitset` 操作数中。

第 4 章

数组和指针

目录

4.1 数组.....	96
4.2 指针的引入.....	100
4.3 C 风格字符串	113
4.4 多维数组	122
小结.....	124
术语.....	125

C++语言提供了两种类似于 `vector` 和迭代器类型的低级复合类型——数组和指针。与 `vector` 类型相似，数组也可以保存某种类型的一组对象；而它们的区别在于，数组的长度是固定的。数组一经创建，就不允许添加新的元素。指针则可以像迭代器一样用于遍历和检查数组中的元素。

现代 C++ 程序应尽量使用 `vector` 和迭代器类型，而避免使用低级的数组和指针。设计良好的程序只有在强调速度时才在类实现的内部使用数组和指针。

数组是 C++ 语言中类似于标准库 `vector` 类型的内置数据结构。与 `vector` 类似，数组也是一种存储单一数据类型对象的容器，其中每个对象都没有单独的名字，而是通过它在数组中的位置对它进行访问。

与 `vector` 类型相比，数组的显著缺陷在于：数组的长度是固定的，而且程序员无法知道一个给定数组的长度。数组没有获取其容量大小的 `size` 操作，也不提供 `push_back` 操作在其中自动添加元素。如果需要更改数组的长度，程序员只能创建一个更大的新数组，然后把原数组的所有元素复制到新数组空间中去。



与使用标准 `vector` 类型的程序相比，依赖于内置数组的程序更容易出错而且难于调试。

在出现标准库之前，C++ 程序大量使用数组保存一组对象。而现代的 C++ 程序则更多地使用 `vector` 来取代数组，数组被严格限制于程序内部使用，只有当性能测试表明使用 `vector` 无法达到必要的速度要求时，才使用数组。然而，在将来一段时间之内，原来依赖于数组的程序仍大量存在，因此，C++ 程序员还是必须掌握数组的使用方法。

4.1 数组

数组是由类型名、标识符和维数组成的复合数据类型（2.5 节），类型名规定了存放在数组中的元素的类型，而维数则指定数组中包含的元素个数。



数组定义中的类型名可以是内置数据类型或类类型，除引用之外，数组元素的类型还可以是任意的复合类型。没有所有元素都是引用的数组。

4.1.1 数组的定义和初始化

数组的维数必须用值大于等于 1 的常量表达式定义（2.7 节）。此常量表达式只能包含整型字面值常量、枚举常量（2.7 节）或者用常量表达式初始化的整型 `const` 对象。非 `const` 变量以及要到运行阶段才知道其值的 `const` 变量都不能用于定义数组的维数。

数组的维数必须在一对方括号 `[]` 内指定：

```
// both buf_size and max_files are const
const unsigned buf_size = 512, max_files = 20;
int staff_size = 27; // nonconst
const unsigned sz = get_size(); // const value not known until run time
char input_buffer[buf_size]; // ok: const variable
string fileTable[max_files + 1]; // ok: constant expression
double salaries[staff_size]; // error: non const variable
int test_scores[get_size()]; // error: non const expression
int vals[sz]; // error: size not known until run time
```

虽然 `staff_size` 是用字面值常量进行初始化，但 `staff_size` 本身是一个非 `const` 对象，只有在运行时才能获得它的值，因此，使用该变量来定义数组维数是非法的。而对于 `sz`，尽管它

是一个 const 对象，但它的值要到运行时调用 get_size 函数后才知道，因此，它也不能用于定义数组维数。另一方面，由于 max_files 是 const 变量，因此表达式

```
max_files+1
```

是常量表达式，编译时即可计算出该表达式的值为 21。

1. 显式初始化数组元素

在定义数组时，可为其元素提供一组用逗号分隔的初值，这些初值用花括号{}括起来，称为初始化列表：

```
const unsigned array_size = 3;
int ia[array_size] = {0, 1, 2};
```

如果没有显式提供元素初值，则数组元素会像普通变量一样初始化（2.3.4 节）：

- 在函数体外定义的内置数组，其元素均初始化为 0；
- 在函数体内定义的内置数组，其元素无初始化；
- 不管数组在哪里定义，如果其元素为类类型，则自动调用该类的默认构造函数进行初始化；如果该类没有默认构造函数，则必须为该数组的元素提供显式初始化。



除非显式地提供元素初值，否则内置类型的局部数组的元素没有初始化。此时，除了给元素赋值外，其他使用这些元素的操作没有定义。

显式初始化的数组不需要指定数组的维数值，编译器会根据列出的元素个数来确定数组的长度：

```
int ia[] = {0, 1, 2}; // an array of dimension 3
```

如果指定了数组维数，那么初始化列表提供的元素个数不能超过维数值。如果维数大于列出的元素初值个数，则只初始化前面的数组元素；剩下的其他元素，若是内置类型则初始化为 0，若是类类型则调用该类的默认构造函数进行初始化。

111

```
const unsigned array_size = 5;
// Equivalent to ia = {0, 1, 2, 0, 0}
// ia[3] and ia[4] default initialized to 0
int ia[array_size] = {0, 1, 2};
// Equivalent to str_arr = {"hi", "bye", "", "", ""}
// str_arr[2] through str_arr[4] default initialized to the empty string
string str_arr[array_size] = {"hi", "bye"};
```

2. 特殊的字符数组

字符数组既可以用一组由花括号括起来、逗号隔开的字符字面值进行初始化，也可以用一个字符串字面值进行初始化。然而，要注意这两种初始化形式并不完全相同，字符串字面值（2.2 节）包含一个额外的空字符（null）用于结束字符串。当使用字符串字面值来初始化创建的新数组时，将在新数组中加入空字符：

```
char ca1[] = {'C', '+', '+'}; // no null
char ca2[] = {'C', '+', '+', '\0'}; // explicit null
char ca3[] = "C++"; // null terminator added automatically
```

ca1 的维数是 3，而 ca2 和 ca3 的维数则是 4。使用一组字符字面值初始化字符数组时，一定要

记得添加结束字符串的空字符。例如，下面的初始化将导致编译时的错误：

```
const char ca3[6] = "Daniel" ;// error: Daniel is 7 elements
```

上述字符串字面值包含了 6 个显式字符，存放该字符串的数组则必须有 7 个元素——6 个用于存储字符字面值，而 1 个用于存放空字符 `null`。

3. 不允许数组直接复制和赋值

与 `vector` 不同，一个数组不能用另外一个数组初始化，也不能将一个数组赋值给另一个数组，这些操作都是非法的：

```
int ia[] = {0, 1, 2}; // ok: array of ints
int ia2[] (ia);       // error: cannot initialize one array with another
int main()
{
    const unsigned array_size = 3;
    int ia3[array_size]; // ok: but elements are uninitialized!

    ia3 = ia;           // error: cannot assign one array to another
    return 0;
}
```



一些编译器允许将数组赋值作为编译器扩展。但是如果希望编写的程序能在不同的编译器上运行，则应该避免使用像数组赋值这类依赖于编译器的非标准功能。

112

习题

习题 4.1 假设 `get_size` 是一个没有参数并返回 `int` 值的函数，下列哪些定义是非法的？为什么？

```
unsigned buf_size = 1024;

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

习题 4.2 下列数组的值是什么？

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

习题 4.3 下列哪些定义是错误的？

```
(a) int ia[7] = {0, 1, 1, 2, 3, 5, 8};
(b) vector<int> ivec = {0, 1, 1, 2, 3, 5, 8};
(c) int ia2[] = ia1;
(d) int ia3[] = ivec ;
```

习题 4.4 如何初始化数组的一部分或全部元素？

习题 4.5 列出使用数组而不是 `vector` 的缺点。

警告：数组的长度是固定的

与 vector 类型不同，数组不提供 `push_back` 或者其他的操作在数组中添加新元素，数组一经定义，就不允许再添加新元素。

如果必须在数组中添加新元素，程序员就必须自己管理内存：要求系统重新分配一个新的内存空间用于存放更大的数组，然后把原数组的所有元素复制到新分配的内存空间中。我们将会在 4.3.1 节学习如何去实现。

4.1.2 数组操作

与 vector 元素一样，数组元素可用下标操作符（3.3.2 节）来访问，数组元素也是从 0 开始计数。对于一个包含 10 个元素的数组，正确的下标值是从 0 到 9，而不是从 1 到 10。

在用下标访问元素时，vector 使用 `vector::size_type` 作为下标的类型，而数组下标的正确类型则是 `size_t`（3.5.2 节）。

113

在下面的例子中，for 循环遍历数组的 10 个元素，并以其下标值作为各个元素的初始值：

```
int main()
{
    const size_t array_size = 10;
    int ia[array_size]; // 10 ints, elements are uninitialized

    // loop through array, assigning value of its index to each element
    for (size_t ix = 0; ix != array_size; ++ix)
        ia[ix] = ix;
    return 0;
}
```

使用类似的循环，可以实现把一个数组复制给另一个数组：

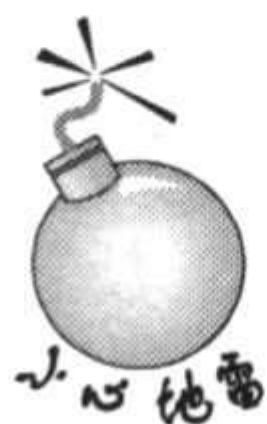
```
int main()
{
    const size_t array_size = 7;
    int ia1[] = { 0, 1, 2, 3, 4, 5, 6 };
    int ia2[array_size]; // local array, elements uninitialized

    // copy elements from ia1 into ia2
    for (size_t ix = 0; ix != array_size; ++ix)
        ia2[ix] = ia1[ix];
    return 0;
}
```

检查数组下标值

正如 string 和 vector 类型，程序员在使用数组时，也必须保证其下标值在正确范围之内，即数组在该下标位置应对应一个元素。

除了程序员自己注意细节，并彻底测试自己的程序之外，没有别的办法可防止数组越界。通过编译并执行的程序仍然存在致命的错误，这并不是不可能的。



导致安全问题的最常见原因是所谓“缓冲区溢出 (buffer overflow)”错误。当我们在编程时没有检查下标，并且引用了越出数组或其他类似数据结构边界的元素时，就会导致这类错误。

习题

习题 4.6 下面的程序段企图将下标值赋给数组的每个元素，其中在下标操作上有一些错误，请指出这些错误。

```
const size_t array_size = 10 ;
int ia[array_size];

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

习题 4.7 编写必要的代码将一个数组赋给另一个数组，然后把这段代码改用 vector 实现。考虑如何将一个 vector 赋给另一个 vector。

习题 4.8 编写程序判断两个数组是否相等，然后编写一段类似的程序比较两个 vector。

习题 4.9 编写程序定义一个有 10 个 int 型元素的数组，并以其在数组中的位置作为各元素的初值。

4.2 指针的引入

vector 的遍历可使用下标或迭代器实现，同理，也可用下标或指针 (pointer) 来遍历数组。指针是指向某种类型对象的复合数据类型，是用于数组的迭代器：指向数组中的一个元素。在指向数组元素的指针上使用解引用操作符 * (dereference operator) 和自增操作符 ++ (increment operator)，与在迭代器上的用法类似。对指针进行解引用操作，可获得该指针所指对象的值。而当指针做自增操作时，则移动指针使其指向数组中的下一个元素。在使用指针编写程序之前，我们需进一步了解一下指针。

114

4.2.1 什么是指针

对初学者来说，指针通常比较难理解。而由指针错误引起的调试问题连富有经验的程序员都感到头疼。然而，指针是大多数 C 程序的重要部分，而且在许多 C++ 程序中仍然受到重用。

指针的概念很简单：指针用于指向对象。与迭代器一样，指针提供对其所指对象的间接访问，只是指针结构更通用一些。与迭代器不同的是，指针用于指向单个对象，而迭代器只能用于访问容器内的元素。

具体来说，指针保存的是另一个对象的地址：

```
string s("hello world");
string *sp = &s; // sp holds the address of s
```

第二条语句定义了一个指向 string 类型的指针 sp，并初始化 sp 使其指向 string 类型的对象 s。*sp 中的 * 操作符表明 sp 是一个指针变量，&s 中的 & 符号是取地址 (address-of) 操作符，当此

操作符用于一个对象上时,返回的是该对象的存储地址。取地址操作符只能用于左值(2.3.1节),因为只有当变量用作左值时,才能取其地址。同样地,由于用于 vector 类型、string 类型或内置数组的下标操作和解引用操作生成左值,因此可对这两种操作的结果做取地址操作,这样即可获取某一特定对象的存储地址。

115

建议: 尽量避免使用指针和数组

指针和数组容易产生不可预料的错误。其中一部分是概念上的问题: 指针用于低级操作,容易产生与繁琐细节相关的(bookkeeping)错误。其他错误则源于使用指针的语法规则,特别是声明指针的语法。

许多有用的程序都可不使用数组或指针实现,现代 C++ 程序采用 vector 类型和迭代器取代一般的数组、采用 string 类型取代 C 风格字符串。

4.2.2 指针的定义和初始化

每个指针都有一个与之关联的数据类型,该数据类型决定了指针所指向的对象的类型。例如,一个 int 型指针只能指向 int 型对象。

1. 指针变量的定义

C++ 语言使用 * 符号把一个标识符声明为指针:

```
vector<int>    *pvec;        // pvec can point to a vector<int>
int            *ip1, *ip2;   // ip1 and ip2 can point to an int
string        *pstring;     // pstring can point to a string
double        *dp;          // dp can point to a double
```



理解指针声明语句时,请从右向左阅读。

从右向左阅读 pstring 变量的定义,可以看到

```
string *pstring;
```

语句把 pstring 定义为一个指向 string 类型对象的指针变量。类似地,语句

```
int *ip1, *ip2; // ip1 and ip2 can point to an int
```

把 ip1 和 ip2 都定义为指向 int 型对象的指针。

在声明语句中,符号 * 可用在指定类型的对象列表的任何位置:

```
double dp, *dp2; // dp2 is a pointer, dp is an object: both type double
```

该语句定义了一个 double 类型的 dp 对象以及一个指向 double 类型对象的指针 dp2。

2. 另一种声明指针的风格

在定义指针变量时,可用空格将符号 * 与其后的标识符分隔开来。下面的写法是合法的:

```
string* ps; // legal but can be misleading
```

也就是说,该语句把 ps 定义为一个指向 string 类型对象的指针。

这种指针声明风格容易引起这样的误解: 把 string* 理解为一种数据类型,认为在同一声明

116 语句中定义的其他变量也是指向 string 类型对象的指针。然而，语句

```
string* ps1, ps2; // ps1 is a pointer to string, ps2 is a string
```

实际上只把 ps1 定义为指针，而 ps2 并非指针，只是一个普通的 string 对象而已。如果需要在 一个声明语句中定义两个指针，必须在每个变量标识符前再加符号*声明：

```
string* ps1, *ps2; // both ps1 and ps2 are pointers to string
```

3. 连续声明多个指针易导致混淆

连续声明同一类型的多个指针有两种通用的声明风格。其中一种风格是一个声明语句只声明 一个变量，此时，符号*紧挨着类型名放置，强调这个声明语句定义的是一个指针：

```
string* ps1;
string* ps2;
```

另一种风格则允许在一条声明语句中声明多个指针，声明时把符号*靠近标识符放置。这种 风格强调对象是一个指针：

```
string *ps1, *ps2;
```

提示

关于指针的声明，不能说哪种声明风格是唯一正确的方式，重要的是选择一种风 格并持续使用。

在本书中，我们将采用第二种声明风格：将符号*紧贴着指针变量名放置。

4. 指针可能的取值

一个有效的指针必然是以下三种状态之一：保存一个特定对象的地址；指向某个对象后面的 另一对象；或者是 0 值。若指针保存 0 值，表明它不指向任何对象。未初始化的指针是无效的， 直到给该指针赋值后，才可使用它。下列定义和赋值都是合法的：

```
int ival = 1024;
int *pi = 0;           // pi initialized to address no object
int *pi2 = &ival;      // pi2 initialized to address of ival
int *pi3;              // ok, but dangerous, pi3 is uninitialized
pi = pi2;              // pi and pi2 address the same object, e.g. ival
pi2 = 0;               // pi2 now addresses no object
```

5. 避免使用未初始化的指针



很多运行时错误都源于使用了未初始化的指针。

就像使用其他没有初始化的变量一样，使用未初始化的指针时的行为 C++ 标准中并没有定义

117 使用未初始化的指针，它几乎总会导致运行时崩溃。然而，导致崩溃的这一原因很难发现。

对大多数的编译器来说，如果使用未初始化的指针，会将指针中存放的不确定值视为地址， 然后操纵该内存地址中存放的位内容。使用未初始化的指针相当于操纵这个不确定地址中存储的 基础数据。因此，在对未初始化的指针进行解引用时，通常会导致程序崩溃。

C++ 语言无法检测指针是否未被初始化，也无法区分有效地址和由指针分配到的存储空间中

存放的二进制位形成的地址。建议程序员在使用之前初始化所有的变量，尤其是指针。



如果可能的话，除非所指向的对象已经存在，否则不要先定义指针，这样可避免定义一个未初始化的指针。

如果必须分开定义指针和其所指向的对象，则将指针初始化为 0。因为编译器可检测出 0 值的指针，程序可判断该指针并未指向一个对象。

6. 指针初始化和赋值操作的约束

对指针进行初始化或赋值只能使用以下四种类型的值：

- (1) 0 值常量表达式 (2.7 节)，例如，在编译时可获得 0 值的整型 `const` 对象或字面值常量 0。
- (2) 类型匹配的对象地址。
- (3) 另一对象之后的下一地址。
- (4) 同类型的另一个有效指针。

把 `int` 型变量赋给指针是非法的，尽管此 `int` 型变量的值可能为 0。但允许把数值 0 或在编译时可获得 0 值的 `const` 量赋给指针：

```
int ival;
int zero = 0;
const int c_ival = 0;
int *pi = ival; // error: pi initialized from int value of ival
pi = zero;      // error: pi assigned int value of zero
pi = c_ival;    // ok: c_ival is a const with compile-time value of 0
pi = 0;         // ok: directly initialize to literal constant 0
```

除了使用数值 0 或在编译时值为 0 的 `const` 量外，还可以使用 C++ 语言从 C 语言中继承下来的预处理器变量 `NULL` (2.9.2 节)，该变量在 `cstdlib` 头文件中定义，其值为 0。如果在代码中使用了这个预处理器变量，则编译时会自动被数值 0 替换。因此，把指针初始化为 `NULL` 等效于初始化为 0 值：

```
// cstdlib #defines NULL to 0
int *pi = NULL; // ok: equivalent to int *pi = 0;
```

正如其他的预处理器变量一样 (2.9.2 节)，不可以使用 `NULL` 这个标识符给自定义的变量命名。



预处理器变量不是在 `std` 命名空间中定义的，因此其名字应为 `NULL`，而非 `std::NULL`。

除了将在 4.2.5 节和 15.3 节介绍的两种例外情况之外，指针只能初始化或赋值为同类型的变量地址或另一指针：

```
double dval;
double *pd = &dval; // ok: initializer is address of a double
double *pd2 = pd;   // ok: initializer is a pointer to double
int *pi = pd;       // error: types of pi and pd differ
pi = &dval;         // error: attempt to assign address of a double to int *
```

由于指针的类型用于确定指针所指对象的类型，因此初始化或赋值时必须保证类型匹配。指针用

于间接访问对象，并基于指针的类型提供可执行的操作，例如，`int` 型指针只能把其指向的对象当作 `int` 型数据来处理，如果该指针确实指向了其他类型（如 `double` 类型）的对象，则在指针上执行的任何操作都有可能出错。

7. `void*` 指针

C++ 提供了一种特殊的指针类型 `void*`，它可以保存任何类型对象的地址：

```
double obj = 3.14;
double *pd = &obj;
// ok: void* can hold the address value of any data pointer type
void *pv = &obj;      // obj can be an object of any type
pv = pd;              // pd can be a pointer to any type
```

`void*` 表明该指针与一地址值相关，但不清楚存储在此地址上的对象的类型。

`void*` 指针只支持几种有限的操作：与另一个指针进行比较；向函数传递 `void*` 指针或从函数返回 `void*` 指针；给另一个 `void*` 指针赋值。不允许使用 `void*` 指针操纵它所指向的对象。我们将在 5.12.4 节讨论如何重新获取存储在 `void*` 指针中的地址。

习题

习题 4.10 下面提供了两种指针声明的形式，解释宁愿使用第一种形式的原因：

```
int *ip; // good practice
int* ip; // legal but misleading
```

习题 4.11 解释下列声明语句，并指出哪些是非法的，为什么？

```
(a) int* ip;
(b) string s, *sp = 0;
(c) int i; double* dp = &i;
(d) int* ip, ip2;
(e) const int i = 0, *p = i;
(f) string *p = NULL;
```

习题 4.12 已知一指针 `p`，你可以确定该指针是否指向一个有效的对象吗？如果可以，如何确定？如果不可以，请说明原因。

习题 4.13 下列代码中，为什么第一个指针的初始化是合法的，而第二个则不合法？

```
int i = 42;
void *p = &i;
long *lp = &i;
```

4.2.3 指针操作

指针提供间接操纵其所指对象的功能。与对迭代器进行解引用操作（3.4 节）一样，对指针进行解引用可访问它所指的对象，`*` 操作符（解引用操作符）将获取指针所指的对象：

```
string s("hello world");
string *sp = &s; // sp holds the address of s
cout <<*sp;      // prints hello world
```

对 `sp` 进行解引用将获得 `s` 的值，然后用输出操作符输出该值，于是最后一条语句输出了 `s` 的内

容 hello world。

1. 生成左值的解引用操作

解引用操作符返回指定对象的左值，利用这个功能可修改指针所指对象的值：

```
*sp = "goodbye"; // contents of s now changed
```

因为 sp 指向 s，所以给 *sp 赋值也就修改了 s 的值。

也可以修改指针 sp 本身的值，使 sp 指向另外一个新对象：

```
string s2 = "some value";
sp = &s2; // sp now points to s2
```

给指针直接赋值即可修改指针的值——不需要对指针进行解引用。

120

关键概念：给指针赋值或通过指针进行赋值

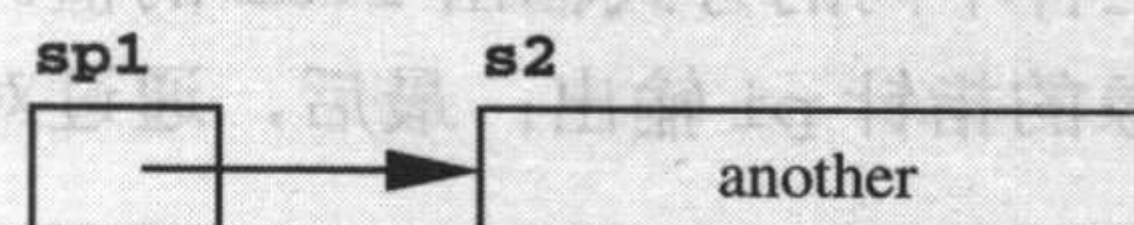
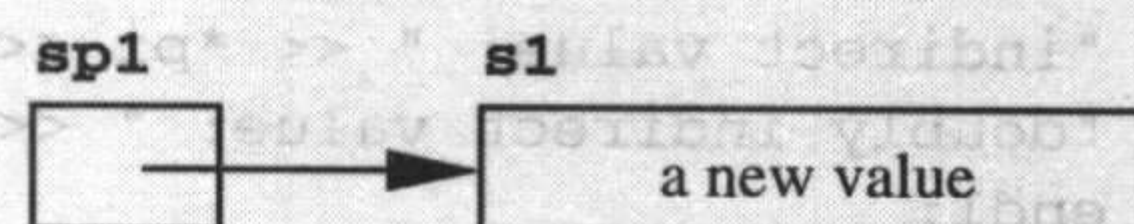
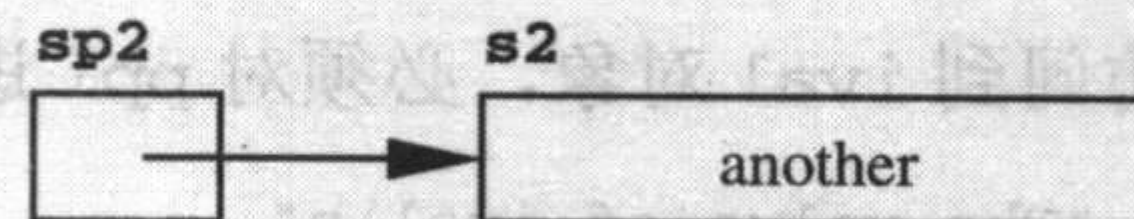
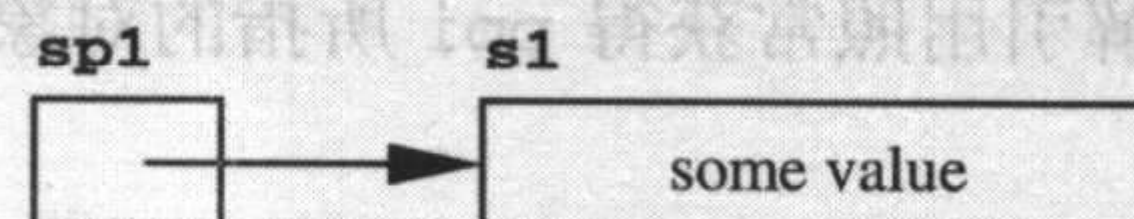
对于初学指针者，给指针赋值和通过指针进行赋值这两种操作的差别确实让人费解。谨记区分的重要方法是：如果对左操作数进行解引用，则修改的是指针所指对象的值；如果没有使用解引用操作，则修改的是指针本身的值。如图所示，帮助理解下列例子：

```
string s1("some value");
string *sp1 = &s1;
```

```
string s2("another");
string *sp2 = &s2;
```

```
// assign through sp1
// value in s1 changed
*sp1 = "a new value";
```

```
// assign to sp1
// sp1 points to a different object
sp1 = sp2;
```



2. 指针和引用的比较

虽然使用引用（reference）和指针都可间接访问另一个值，但它们之间有两个重要区别。第一个区别在于引用总是指向某个对象：定义引用时没有初始化是错误的。第二个重要区别则是赋值行为的差异：给引用赋值修改的是该引用所关联的对象的值，而并不是使引用与另一个对象关联。引用一经初始化，就始终指向同一个特定对象（这就是为什么引用必须在定义时初始化的原因）。

考虑以下两个程序段。第一个程序段将一个指针赋给另一指针：

```
int ival = 1024, ival2 = 2048;
int *pi = &ival, *pi2 = &ival2;
pi = pi2; // pi now points to ival2
```

赋值结束后，pi 所指向的 ival 对象值保持不变，赋值操作修改了 pi 指针的值，使其指向另一

个不同的对象。现在考虑另一段相似的程序，使用两个引用赋值：

```
int &ri = ival, &ri2 = ival2;
ri = ri2;    // assigns ival2 to ival
```

这个赋值操作修改了 `ri` 引用的值 `ival` 对象，而并非引用本身。赋值后，这两个引用还是分别指向原来关联的对象，此时这两个对象的值相等。

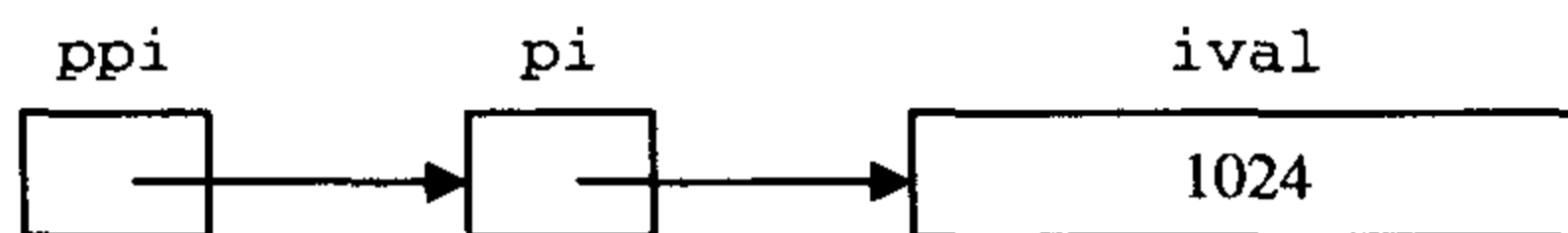
121

3. 指向指针的指针

指针本身也是可用指针指向的内存对象。指针占用内存空间存放其值，因此指针的存储地址可存放在指针中。下面程序段：

```
int ival = 1024;
int *pi = &ival;    // pi points to an int
int **ppi = &pi;    // ppi points to a pointer to int
```

定义了指向指针的指针。C++使用`**`操作符指派一个指针指向另一指针。这些对象可表示为：



对 `ppi` 进行解引用照常获得 `ppi` 所指的对象，在本例中，所获得的对象是指向 `int` 型变量的指针 `pi`：

```
int *pi2 = *ppi;    // ppi points to a pointer
```

为了真正地访问到 `ival` 对象，必须对 `ppi` 进行两次解引用：

```
cout << "The value of ival\n"
    << "direct value: " << ival << "\n"
    << "indirect value: " << *pi << "\n"
    << "doubly indirect value: " << **ppi
    << endl;
```

这段程序用三种不同的方式输出 `ival` 的值。首先，采用直接引用变量的方式输出；然后使用指向 `int` 型对象的指针 `pi` 输出；最后，通过对 `ppi` 进行两次解引用获得 `ival` 的特定值。

习题

习题 4.14 编写代码修改指针的值；然后再编写代码修改指针所指对象的值。

习题 4.15 解释指针和引用的主要区别。

习题 4.16 下列程序段实现什么功能？

```
int i = 42, j = 1024;
int *p1 = &i, *p2 = &j;
*p2 = *p1 * *p2;
*p1 *= *p1;
```

4.2.4 使用指针访问数组元素

C++语言中，指针和数组密切相关。特别是在表达式中使用数组名时，该名字会自动转换为指向数组第一个元素的指针：

122

```
int ia[] = {0, 2, 4, 6, 8};
int *ip = ia;    // ip points to ia[0]
```

如果希望使指针指向数组中的另一个元素，则可使用下标操作符给某个元素定位，然后用取地址操作符&获取该元素的存储地址：

```
ip = &ia[4];    // ip points to last element in ia
```

1. 指针的算术操作

与其使用下标操作，倒不如通过指针的算术操作（pointer arithmetic）来获取指定内容的存储地址。指针的算术操作和迭代器的算术操作（3.4.1 节）以相同的方式实现（也具有相同的约束）。使用指针的算术操作在指向数组某个元素的指针上加上（或减去）一个整型数值，就可以计算出指向数组另一元素的指针值：

```
ip = ia;          // ok: ip points to ia[0]
int *ip2 = ip + 4; // ok: ip2 points to ia[4], the last element in ia
```

在指针 ip 上加 4 得到一个新的指针，指向数组中 ip 当前指向的元素后的第 4 个元素。

通常，在指针上加上（或减去）一个整型数值 n 等效于获得一个新指针，该新指针指向指针原来指向的元素之后（或之前）的第 n 个元素。



指针的算术操作只有在原指针和计算出来的新指针都指向同一个数组的元素，或指向该数组存储空间的下单元时才是合法的。如果指针指向一对象，我们还可以在指针上加 1 从而获取指向相邻的下一个对象的指针。

假设数组 ia 只有 4 个元素，则在 ia 上加 10 是错误的：

```
// error: ia has only 4 elements, ia + 10 is an invalid address
int *ip3 = ia + 10;
```

只要两个指针指向同一数组或有一个指向该数组末端的下一单元，C++还支持对这两个指针做减法操作：

```
ptrdiff_t n = ip2 - ip; // ok: distance between the pointers
```

结果是 4，这两个指针所指向的元素间隔为 4 个对象。两个指针减法操作的结果是标准库类型（library type）`ptrdiff_t` 的数据。与 `size_t` 类型一样，`ptrdiff_t` 也是一种与机器相关的类型，在 `cstdint` 头文件中定义。`size_t` 是 unsigned 类型，而 `ptrdiff_t` 则是 signed 整型。

这两种类型的差别体现了它们各自的用途：`size_t` 类型用于指明数组长度，它必须是一个正数；`ptrdiff_t` 类型则应保证足以存放同一数组中两个指针之间的差距，它有可能是负数。例如，ip 减去 ip2，结果为 -4。

允许在指针上加减 0，使指针保持不变。更有趣的是，如果一指针具有 0 值（空指针），则在该指针上加 0 仍然是合法的，结果得到另一个值为 0 的指针。也可以对两个空指针做减法操作，得到的结果仍是 0。

2. 解引用和指针算术操作之间的相互作用

在指针上加一个整型数值，其结果仍然是指针。允许在这个结果上直接进行解引用操作，而不必先把它赋给一个新指针：

```
int last = *(ia + 4); // ok: initializes last to 8, the value of ia[4]
```

这个表达式计算出 `ia` 所指向元素后面的第 4 个元素的地址，然后对该地址进行解引用操作，等价于 `ia[4]`。



加法操作两边用圆括号括起来是必要的。如果写为：

```
last = *ia + 4; // ok: last = 4, equivalent to ia[0]+4
```

意味着对 `ia` 进行解引用，获得 `ia` 所指元素的值 `ia[0]`，然后加 4。

由于加法操作和解引用操作的优先级 (precedence) 不同，上述表达式中的圆括号是必要的。我们将在 5.10.1 节讨论操作符的优先级。简单地说，优先级决定了有多个操作符的表达式如何对操作数分组。解引用操作符的优先级比加法操作符高。

与低优先级的操作符相比，优先级高的操作符的操作数先被组合起来操作。如果没有圆括号，解引用操作符的操作数是 `ia`，该表达式先对 `ia` 解引用，获得 `ia` 数组中的第一个元素，并将该值与 4 相加。

如果表达式加上圆括号，则不管一般的优先级规则，将 `(ia + 4)` 作为单个操作数，这是 `ia` 所指向的元素后面第 4 个元素的地址，然后对这个新地址进行解引用。

3. 下标和指针

我们已经看到，在表达式中使用数组名时，实际上使用的是指向数组第一个元素的指针。这种用法涉及很多方面，当它们出现时我们会逐一指出来。

其中一个重要的应用是使用下标访问数组时，实际上是使用下标访问指针：

```
int ia[] = {0, 2, 4, 6, 8};
int i = ia[0]; // ia points to the first element in ia
```

`ia[0]` 是一个使用数组名的表达式。在使用下标访问数组时，实际上是对指向数组元素的指针做

124 下标操作。只要指针指向数组元素，就可以对它进行下标操作：

```
int *p = &ia[2]; // ok: p points to the element indexed by 2
int j = p[1];    // ok: p[1] equivalent to *(p + 1),
                // p[1] is the same element as ia[3]
int k = p[-2];   // ok: p[-2] is the same element as ia[0]
```

4. 计算数组的超出末端指针

`vector` 类型提供的 `end` 操作将返回指向超出 `vector` 末端位置的一个迭代器。这个迭代器常用作哨兵，来控制处理 `vector` 中元素的循环。类似地，可以计算数组的超出末端指针的值：

```
const size_t arr_size = 5;
int arr[arr_size] = {1, 2, 3, 4, 5};
int *p = arr; // ok: p points to arr[0]
int *p2 = p + arr_size; // ok: p2 points one past the end of arr
// use caution -- do not dereference!
```

本例中，`p` 指向数组 `arr` 的第一个元素，在指针 `p` 上加数组长度即可计算出数组 `arr` 的超出末端指针。`p` 加 5 即得 `p` 所指向的元素后面的第五个 `int` 元素的地址——换句话说，`p+5` 指向数组的超出末端的位置。



C++允许计算数组或对象的超出末端的地址，但不允许对此地址进行解引用操作。而计算数组超出末端位置之后或数组首地址之前的地址都是不合法的。

计算并存储在 p2 中的地址，与在 vector 上做 end 操作所返回的迭代器具有相同的功能。由 end 返回的迭代器标志了该 vector 对象的“超出末端位置”，不能进行解引用运算，但是可将它与别的迭代器比较，从而判断是否已经处理完 vector 中所有的元素。同理，p2 也只能用来与其他指针比较，或者用做指针算术操作表达式的操作数。对 p2 进行解引用将得到无效值。对大多数的编译器来说，会把对 p2 进行解引用的结果（恰好存储在 arr 数组的最后一个元素后面的内存中的二进制位）视为一个 int 型数据。

5. 输出数组元素

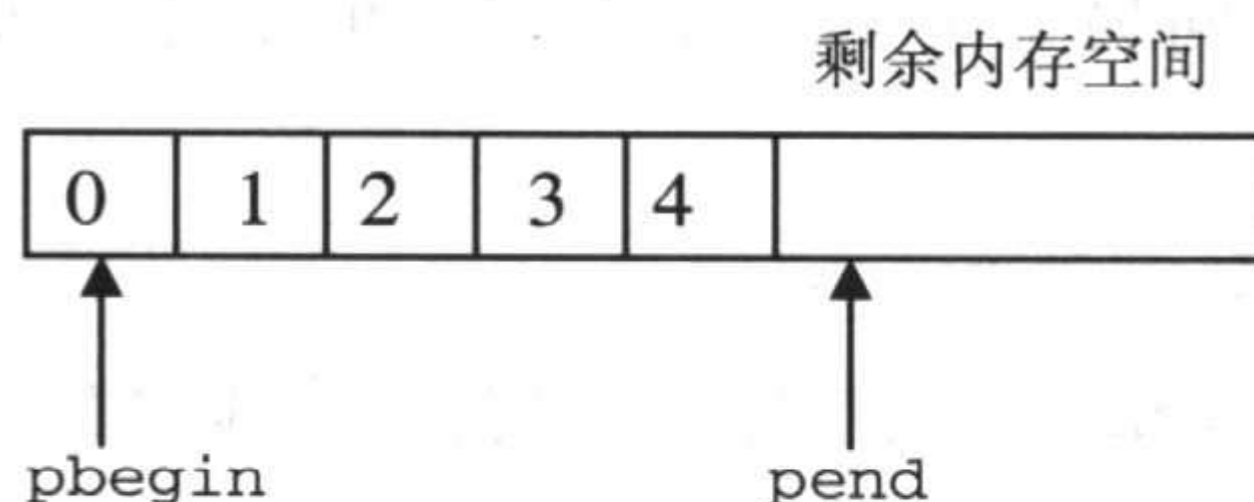
用指针编写以下程序：

```
const size_t arr_sz = 5;
int int_arr[arr_sz] = { 0, 1, 2, 3, 4 };
// pbegin points to first element, pend points just after the last
for (int *pbegin = int_arr, *pend = int_arr + arr_sz;
     pbegin != pend; ++pbegin)
    cout << *pbegin << ' '; // print the current element
```

这段程序使用了一个我们以前没有用过的 for 循环性质：只要定义多个变量具有相同的类型，就可以在 for 循环的初始化语句（init-statement，1.4.2 节）中同时定义它们。本例在初始化语句中定义了两个 int 型指针 pbegin 和 pend。

125

C++允许使用指针遍历数组。和其他内置类型一样，数组也没有成员函数。因此，数组不提供 begin 和 end 操作，程序员只能自己给指针定位，使之分别标志数组的起始位置和超出末端位置。可在初始化中实现这两个指针的定位：初始化指针 pbegin 指向 int_arr 数组的第一个元素，而指针 pend 则指向该数组的超出末端的位置：



指针 pend 是标志 for 循环结束的哨兵。for 循环的每次迭代都会使 pbegin 递增 1 以指向数组的下一个元素。第一次执行 for 循环时，pbegin 指向数组中的第一个元素；第二次循环，指向第二个元素；这样依次类推。当处理完数组的最后一个元素后，pbegin 再加 1 则与 pend 值相等，表示整个数组已遍历完毕。

6. 指针是数组的迭代器

聪明的读者可能已经注意到这段程序与 3.4 节的一段程序非常相像，该程序使用下面的循环遍历并输出一个 string 类型的 vector 的内容：

```
// equivalent loop using iterators to reset all the elements in ivec to 0
for (vector<int>::iterator iter = ivec.begin();
     iter != ivec.end(); ++iter)
```

```
*iter = 0; // set element to which iter refers to 0
```

这段程序使用迭代器的方式就像上个程序使用指针实现输出数组内容一样。指针和迭代器的这个相似之处并不是巧合。实际上，内置数组类型具有标准库容器的许多性质，与数组联合使用的指针本身就是迭代器。在第二部分中，我们还会详细介绍容器和迭代器类型。

习题

习题 4.17 已知 p1 和 p2 指向同一个数组中的元素，下面语句实现什么功能？

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

当 p1 和 p2 具有什么值时这个语句是非法的？

习题 4.18 编写程序，使用指针把一个 int 型数组的所有元素设置为 0。

4.2.5 指针和 const 限定符

2.4 节介绍了指针和 const 限定符之间的两种交互类型：指向 const 对象的指针和 const 指针。我们在本节中详细讨论这两类指针。

1. 指向 const 对象的指针

126

到目前为止，我们使用指针来修改其所指对象的值。但是如果指针指向 const 对象，则不允许用指针来改变其所指的 const 值。为了保证这个特性，C++ 语言强制要求指向 const 对象的指针也必须具有 const 特性：

```
const double *cptr ; // cptr may point to a double that is const
```

这里的 cptr 是一个指向 double 类型 const 对象的指针，const 限定了 cptr 指针所指向的对象类型，而并非 cptr 本身。也就是说，cptr 本身并不是 const。在定义时不需要对它进行初始化，如果需要的话，允许给 cptr 重新赋值，使其指向另一个 const 对象。但不能通过 cptr 修改其所指对象的值：

```
*cptr = 42 ; // error: *cptr might be const
```

把一个 const 对象的地址赋给一个普通的、非 const 对象的指针也会导致编译时的错误：

```
const double pi = 3.14;
double *ptr = &pi; // error: ptr is a plain pointer
const double *cptr = &pi; // ok: cptr is a pointer to const
```

不能使用 void* 指针（4.2.2 节）保存 const 对象的地址，而必须使用 const void* 类型的指针保存 const 对象的地址：

```
const int universe = 42;
const void *cpv = &universe; // ok: cpv is const
void *pv = &universe; // error: universe is const
```

允许把非 const 对象的地址赋给指向 const 对象的指针，例如：

```
double dval = 3.14; // dval is a double; its value can be changed
cptr = &dval; // ok: but can't change dval through cptr
```

尽管 `dval` 不是 `const` 对象，但任何企图通过指针 `cptr` 修改其值的行为都会导致编译时的错误。`cptr` 一经定义，就不允许修改其所指对象的值。如果该指针恰好指向非 `const` 对象时，同样必须遵循这个规则。

127



不能使用指向 `const` 对象的指针修改基础对象，然而如果该指针指向的是一个非 `const` 对象，可用其他方法修改其所指的对象。

事实是，可以修改 `const` 指针所指向的值，这一点常常容易引起误会。考虑：

```
dval = 3.14159;           // dval is not const
*cptr = 3.14159;          // error: cptr is a pointer to const
double *ptr = &dval;       // ok: ptr points at non-const double
*ptr = 2.72;               // ok: ptr is plain pointer
cout << *cptr;             // ok: prints 2.72
```

在此例题中，指向 `const` 的指针 `cptr` 实际上指向了一个非 `const` 对象。尽管它所指的对象并非 `const`，但仍不能使用 `cptr` 修改该对象的值。本质上来说，由于没有方法分辨 `cptr` 所指的对象是否为 `const`，系统会把它所指的所有对象都视为 `const`。

如果指向 `const` 的指针所指的对象并非 `const`，则可直接给该对象赋值或间接地利用普通的非 `const` 指针修改其值：毕竟这个值不是 `const`。重要的是要记住：不能保证指向 `const` 的指针所指对象的值一定不可修改。



如果把指向 `const` 的指针理解为“自以为指向 `const` 的指针”，这可能会对理解有所帮助。

在实际的程序中，指向 `const` 的指针常用作函数的形参。将形参定义为指向 `const` 的指针，以此确保传递给函数的实际对象在函数中不会因为形参而被修改。

2. `const` 指针

除指向 `const` 对象的指针外，C++语言还提供了 `const` 指针——本身的值不能修改：

```
int errNumb = 0;
int *const curErr = &errNumb; // curErr is a constant pointer
```

我们可以从右向左把上述定义语句读作“`curErr`是指向 `int` 型对象的 `const` 指针”。与其他 `const` 量一样，`const` 指针的值不能修改，这就意味着不能使 `curErr` 指向其他对象。任何企图给 `const` 指针赋值的行为（即使给 `curErr` 赋回同样的值）都会导致编译时的错误：

```
curErr = curErr; // error: curErr is const
```

与任何 `const` 量一样，`const` 指针也必须在定义时初始化。

128

指针本身是 `const` 的事实并没有说明是否能使用该指针修改它所指向对象的值。指针所指对象的值能否修改完全取决于该对象的类型。例如，`curErr` 指向一个普通的非常量 `int` 型对象 `errNumb`，则可使用 `curErr` 修改该对象的值：

```
if (*curErr) {
    errorHandler();
    *curErr = 0; // ok: reset value of the object to which curErr is bound
}
```

3. 指向 **const** 对象的 **const** 指针

还可以如下定义指向 **const** 对象的 **const** 指针：

```
const double pi = 3.14159;
// pi_ptr is const and points to a const object
const double *const pi_ptr = &pi;
```

本例中，既不能修改 `pi_ptr` 所指向对象的值，也不允许修改该指针的指向（即 `pi_ptr` 中存放的地址值）。可从右向左阅读上述声明语句：“`pi_ptr` 首先是一个 **const** 指针，指向 `double` 类型的 **const** 对象”。

4. 指针和 **typedef**

在 **typedef**（2.6 节）中使用指针往往会带来意外的结果。下面是一个几乎所有人刚开始时都会答错的问题。假设给出以下语句：

```
typedef string *pstring;
const pstring cstr;
```

请问 `cstr` 变量是什么类型？简单的回答是 `const pstring` 类型的指针。进一步问：`const pstring` 指针所表示的真实类型是什么？很多人都认为真正的类型是：

```
const string *cstr; // wrong interpretation of const pstring cstr
```

也就是说，`const pstring` 是一种指针，指向 `string` 类型的 **const** 对象，但这是错误的。

错误的原因在于将 **typedef** 当做文本扩展了。声明 `const pstring` 时，`const` 修饰的是 `pstring` 的类型，这是一个指针。因此，该声明语句应该是把 `cstr` 定义为指向 `string` 类型对象的 **const** 指针，这个定义等价于：

```
// cstr is a const pointer to string
string *const cstr; // equivalent to const pstring cstr
```

129

建议：理解复杂的 **const** 类型的声明

阅读 **const** 声明语句产生的部分问题，源于 **const** 限定符既可以放在类型前也可以放在类型后：

```
string const s1; // s1 and s2 have same type,
const string s2; // they're both strings that are const
```

用 **typedef** 写 **const** 类型定义时，**const** 限定符加在类型名前面容易引起对所定义的真正类型的误解：

```
string s;
typedef string *pstring;
const pstring cstr1 = &s; // written this way the type is obscured
pstring const cstr2 = &s; // all three declarations are the same type
string *const cstr3 = &s; // they're all const pointers to string
```

把 **const** 放在类型 `pstring` 之后，然后从右向左阅读该声明语句就会非常清楚地知道 `cstr2` 是 `const pstring` 类型，即指向 `string` 对象的 **const** 指针。

不幸的是，大多数人在阅读 C++ 程序时都习惯看到 **const** 放在类型前面。于是为了遵照惯例，只好建议编程时把 **const** 放在类型前面。但是，把声明语句重写为置 **const** 于类型之后更便于理解。

习题

习题 4.19 解释下列 5 个定义的含义, 指出其中哪些定义是非法的:

- (a) `int i;`
- (b) `const int ic;`
- (c) `const int *pic;`
- (d) `int *const cpi;`
- (e) `const int *const cpic;`

习题 4.20 下列哪些初始化是合法的? 为什么?

- (a) `int i = -1;`
- (b) `const int ic = i ;`
- (c) `const int *pic = ⁣`
- (d) `int *const cpi = ⁣`
- (e) `const int *const cpic = &ic ;`

习题 4.21 根据上述定义, 下列哪些赋值运算是合法的? 为什么?

- (a) `i = ic;`
- (b) `pic = ⁣`
- (c) `cpi = pic;`
- (d) `pic = cpic;`
- (e) `cpic = ⁣`
- (f) `ic = *cpic;`

4.3 C 风格字符串



尽管 C++ 支持 C 风格字符串, 但不应该在 C++ 程序中使用这个类型。C 风格字符串常常带来许多错误, 是导致大量安全问题的根源。

在 2.2 节中我们第一次使用了字符串字面值, 并了解字符串字面值的类型是字符常量的数组, 现在可以更明确地认识到: 字符串字面值的类型就是 `const char` 类型的数组。C++ 从 C 语言继承下来的一种通用结构是 C 风格字符串 (C-style character string), 而字符串字面值就是该类型的实例。实际上, C 风格字符串既不能确切地归结为 C 语言的类型, 也不能归结为 C++ 语言的类型, 而是以空字符 `null` 结束的字符数组:

```
char ca1[] = {'C', '+', '+'};           // no null, not C-style string
char ca2[] = {'C', '+', '+', '\0'};    // explicit null
char ca3[] = "C++";                    // null terminator added automatically
const char *cp = "C++";                // null terminator added automatically
char *cp1 = ca1;                       // points to first element of a array, but not C-style string
char *cp2 = ca2;                       // points to first element of a null-terminated char array
```

`ca1` 和 `cp1` 都不是 C 风格字符串: `ca1` 是一个不带结束符 `null` 的字符数组, 而指针 `cp1` 指向 `ca1`, 因此, 它指向的并不是以 `null` 结束的数组。其他的声明则都是 C 风格字符串, 数组的名字即是指向该数组第一个元素的指针。于是, `ca2` 和 `ca3` 分别是指向各自数组第一个元素的指针。

1. C 风格字符串的使用

C++ 语言通过 `(const) char*` 类型的指针来操纵 C 风格字符串。一般来说, 我们使用指针的算术操作来遍历 C 风格字符串, 每次对指针进行测试并递增 1, 直到到达结束符 `null` 为止:

```

const char *cp = "some value";
while (*cp) {
    // do something to *cp
    ++cp;
}

```

while 语句的循环条件是对 `const char*` 类型的指针 `cp` 进行解引用，并判断 `cp` 当前指向的字符是 `true` 值还是 `false` 值。真值表明这是除 `null` 外的任意字符，则继续循环直到 `cp` 指向结束字符数组的 `null` 时，循环结束。while 循环体做完必要的处理后，`cp` 加 1，向下移动指针指向数组中的下一个字符。



如果 `cp` 所指向的字符数组没有 `null` 结束符，则此循环将会失败。这时，循环会从 `cp` 指向的位置开始读数，直到遇到内存中某处 `null` 结束符为止。

131

2. C 风格字符串的标准库函数

表 4-1 列出了 C 语言标准库提供的一系列处理 C 风格字符串的库函数。要使用这些标准库函数，必须包含相应的 C 头文件：

```
#include <cstring>
```

`cstring` 是 `string.h` 头文件的 C++ 版本，而 `string.h` 则是 C 语言提供的标准库。



这些标准库函数不会检查其字符串参数。

传递给这些标准库函数例程的指针必须具有非零值，并且指向以 `null` 结束的字符数组中的第一个元素。其中一些标准库函数会修改传递给它的字符串，这些函数将假定它们所修改的字符串具有足够大的空间接收本函数新生成的字符，程序员必须确保目标字符串必须足够大。

表 4-1 操纵 C 风格字符串的标准库函数

<code>strlen(s)</code>	返回 <code>s</code> 的长度，不包括字符串结束符 <code>null</code>
<code>strcmp(s1, s2)</code>	比较两个字符串 <code>s1</code> 和 <code>s2</code> 是否相同。若 <code>s1</code> 与 <code>s2</code> 相等，返回 0；若 <code>s1</code> 大于 <code>s2</code> ，返回正数；若 <code>s1</code> 小于 <code>s2</code> ，则返回负数
<code>strcat(s1, s2)</code>	将字符串 <code>s2</code> 连接到 <code>s1</code> 后，并返回 <code>s1</code>
<code>strcpy(s1, s2)</code>	将 <code>s2</code> 复制给 <code>s1</code> ，并返回 <code>s1</code>
<code>strncat(s1, s2, n)</code>	将 <code>s2</code> 的前 <code>n</code> 个字符连接到 <code>s1</code> 后面，并返回 <code>s1</code>
<code>strncpy(s1, s2, n)</code>	将 <code>s2</code> 的前 <code>n</code> 个字符复制给 <code>s1</code> ，并返回 <code>s1</code>

C++ 语言提供普通的关系操作符实现标准库类型 `string` 的对象的比较。这些操作符也可用于比较指向 C 风格字符串的指针，但效果却很不相同：实际上，此时比较的是指针上存放的地址值，而并非它们所指向的字符串：

```
if (cp1 < cp2) // compares addresses, not the values pointed to
```

如果 `cp1` 和 `cp2` 指向同一数组中的元素（或该数组的溢出位置），上述表达式等效于比较在 `cp1` 和 `cp2` 中存放的地址；如果这两个指针指向不同的数组，则该表达式实现的比较没有定义。

字符串的比较和比较结果的解释都须使用标准库函数 `strcmp` 进行：

```
const char *cp1 = "A string example";
const char *cp2 = "A different string";
int i = strcmp(cp1, cp2);      // i is positive
i = strcmp(cp2, cp1);        // i is negative
i = strcmp(cp1, cp1);        // i is zero
```

132

标准库函数 `strcmp` 有 3 种可能的返回值：若两个字符串相等，则返回 0 值；若第一个字符串大于第二个字符串，则返回正数，否则返回负数。

3. 永远不要忘记字符串结束符 `null`

在使用处理 C 风格字符串的标准库函数时，牢记字符串必须以结束符 `null` 结束：

```
char ca[] = {'C', '+', '+'}; // not null-terminated
cout << strlen(ca) << endl; // disaster: ca isn't null-terminated
```

在这个例题中，`ca` 是一个没有 `null` 结束符的字符数组，则计算的结果不可预料。标准库函数 `strlen` 总是假定其参数字符串以 `null` 字符结束，当调用该标准库函数时，系统将会从实参 `ca` 指向的内存空间开始一直搜索结束符，直到恰好遇到 `null` 为止。`strlen` 返回这一段内存空间中总共有多少个字符，无论如何这个数值不可能是正确的。

4. 调用者必须确保目标字符串具有足够的大小

传递给标准库函数 `strcat` 和 `strcpy` 的第一个实参数组必须具有足够大的空间存放新生成的字符串。以下代码虽然演示了一种通常的用法，但是却有潜在的严重错误：

```
// Dangerous: What happens if we miscalculate the size of largeStr?
char largeStr[16 + 18 + 2]; // will hold cp1 a space and cp2
strcpy(largeStr, cp1);      // copies cp1 into largeStr
strcat(largeStr, " ");      // adds a space at end of largeStr
strcat(largeStr, cp2);      // concatenates cp2 to largeStr
// prints A string example A different string
cout << largeStr << endl;
```

问题在于我们经常会算错 `largeStr` 需要的大小。同样地，如果 `cp1` 或 `cp2` 所指向的字符串大小发生了变化，`largeStr` 所需要的大小则会计算错误。不幸的是，类似于上述代码的程序应用非常广泛，这类程序往往容易出错，并导致严重的安全漏洞。

5. 使用 `strn` 函数处理 C 风格字符串

如果必须使用 C 风格字符串，则使用标准库函数 `strncat` 和 `strncpy` 比 `strcat` 和 `strcpy` 函数更安全：

```
char largeStr[16 + 18 + 2]; // to hold cp1 a space and cp2
strncpy(largeStr, cp1, 17); // size to copy includes the null
strncat(largeStr, " ", 2);  // pedantic, but a good habit
strncat(largeStr, cp2, 19); // adds at most 18 characters, plus a null
```

使用标准库函数 `strncat` 和 `strncpy` 的诀窍在于可以适当地控制复制字符的个数。特别是在复制和串连字符串时，一定要时刻记住算上结束符 `null`。在定义字符串时要切记预留存放 `null` 字符

133

的空间，因为每次调用标准库函数后都必须以此结束字符串 `largeStr`。让我们详细分析一下这些标准库函数的调用：

- 调用 `strncpy` 时，要求复制 17 个字符：字符串 `cp1` 中所有字符，加上结束符 `null`。留下存储结束符 `null` 的空间是必要的，这样 `largeStr` 才可以正确地结束。调用 `strncpy` 后，字符串 `largeStr` 的长度 `strlen` 值是 16。记住：标准库函数 `strlen` 用于计算 C 风格字符串中的字符个数，不包括 `null` 结束符。
- 调用 `strncat` 时，要求复制 2 个字符：一个空格和结束该字符串字面值的 `null`。调用结束后，字符串 `largeStr` 的长度是 17，原来用于结束 `largeStr` 的 `null` 被新添加的空格覆盖了，然后在空格后面写入新的结束符 `null`。
- 第二次调用 `strncat` 串接 `cp2` 时，要求复制 `cp2` 中所有字符，包括字符串结束符 `null`。调用结束后，字符串 `largeStr` 的长度是 35：`cp1` 的 16 个字符和 `cp2` 的 18 个字符，再加上分隔这两个字符串的一个空格。

整个过程中，存储 `largeStr` 的数组大小始终保持为 36（包括结束符）。

只要可以正确计算出 `size` 实参的值，使用 `strn` 版本要比没有 `size` 参数的简化版本更安全。但是，如果要向目标数组复制或串接比其 `size` 更多的字符，数组溢出的现象仍然会发生。如果要复制或串接的字符串比实际要复制或串接的 `size` 大，我们会不经意地把新生成的字符串截短了。截短字符串比数组溢出要安全，但这仍是错误的。

6. 尽可能使用标准库类型 `string`

如果使用 C++ 标准库类型 `string`，则不存在上述问题：

```
string largeStr = cp1; // initialize largeStr as a copy of cp1
largeStr += " ";      // add space at end of largeStr
largeStr += cp2;       // concatenate cp2 onto end of largeStr
```

此时，标准库负责处理所有的内存管理问题，我们不必再担心每一次修改字符串时涉及到的大小问题。



对大部分的应用而言，使用标准库类型 `string`，除了增强安全性外，效率也提高了，因此应该尽量避免使用 C 风格字符串。

习题

习题 4.22 解释下列两个 `while` 循环的差别：

```
const char *cp = "hello";
int cnt;
while (cp) { ++cnt; ++cp; }
while (*cp) { ++cnt; ++cp; }
```

习题 4.23 下列程序实现什么功能？

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

```
    ++cp;
}
```

习题 4.24 解释 strcpy 和 strncpy 的差别在哪里, 各自的优缺点是什么?

习题 4.25 编写程序比较两个 string 类型的字符串, 然后编写另一个程序比较两个 C 风格字符串的值。

习题 4.26 编写程序从标准输入设备读入一个 string 类型的字符串。考虑如何编程实现从标准输入设备读入一个 C 风格字符串。

4.3.1 创建动态数组

数组类型的变量有三个重要的限制: 数组长度固定不变, 在编译时必须知道其长度, 数组只在定义它的块语句内存在。实际的程序往往不能忍受这样的限制——它们需要在运行时动态地分配数组。虽然数组长度是固定的, 但动态分配的数组不必在编译时知道其长度, 可以(通常也是)在运行时才确定数组长度。与数组变量不同, 动态分配的数组将一直存在, 直到程序显式释放它为止。

134

每一个程序在执行时都占用一块可用的内存空间, 用于存放动态分配的对象, 此内存空间称为程序的自由存储区 (free store) 或堆 (heap)。C 语言程序使用一对标准库函数 malloc 和 free 在自由存储区中分配存储空间, 而 C++ 语言则使用 new 和 delete 表达式实现相同的功能。

1. 动态数组的定义

数组变量通过指定类型、数组名和维数来定义。而动态分配数组时, 只需指定类型和数组长度, 不必为数组对象命名, new 表达式返回指向新分配数组的第一个元素的指针:

```
int *pia = new int[10]; // array of 10 uninitialized ints
```

此 new 表达式分配了一个含有 10 个 int 型元素的数组, 并返回指向该数组第一个元素的指针, 此返回值初始化了指针 pia。

new 表达式需要指定指针类型以及在方括号中给出的数组维数, 该维数可以是任意的复杂表达式。创建数组后, new 将返回指向数组第一个元素的指针。在自由存储区中创建的数组对象是没有名字的, 程序员只能通过其地址间接地访问堆中的对象。

135

2. 初始化动态分配的数组

动态分配数组时, 如果数组元素具有类类型, 将使用该类的默认构造函数 (2.3.4 节) 实现初始化; 如果数组元素是内置类型, 则无初始化:

```
string *psa = new string[10]; // array of 10 empty strings
int *pia = new int[10];       // array of 10 uninitialized ints
```

这两个 new 表达式都分配了含有 10 个对象的数组。其中第一个数组是 string 类型, 分配了保存对象的内存空间后, 将调用 string 类型的默认构造函数依次初始化数组中的每个元素。第二个数组则具有内置类型的元素, 分配了存储 10 个 int 对象的内存空间, 但这些元素没有初始化。

也可使用跟在数组长度后面的一对空圆括号, 对数组元素做值初始化 (3.3.1 节):

```
int *pia2 = new int[10](); // array of 10 uninitialized ints
```

圆括号要求编译器对数组做值初始化，在本例中即把数组元素都设置为 0。



对于动态分配的数组，其元素只能初始化为元素类型的默认值，而不能像数组变量一样，用初始化列表为数组元素提供各不相同的初值。

3. const 对象的动态数组

如果我们在自由存储区中创建的数组存储了内置类型的 const 对象，则必须为这个数组提供初始化：因为数组元素都是 const 对象，无法赋值。实现这个要求的唯一方法是对数组做值初始化：

```
// error: uninitialized const array
const int *pci_bad = new const int[100];
// ok: value-initialized const array
const int *pci_ok = new const int[100]();
```

C++ 允许定义类类型的 const 数组，但该类类型必须提供默认构造函数：

```
// ok: array of 100 empty strings
const string *pcs = new const string[100];
```

在这里，将使用 string 类的默认构造函数初始化数组元素。

当然，已创建的常量元素不允许修改——因此这样的数组实际上用处不大。

4. 允许动态分配空数组

之所以要动态分配数组，往往是由于编译时并不知道数组的长度。我们可以编写如下代码

```
size_t n = get_size(); // get_size returns number of elements needed
int* p = new int[n];
for (int* q = p; q != p + n; ++q)
    /* process the array */ ;
```

计算数组长度，然后创建和处理该数组。

有趣的是，如果 get_size 返回 0 则会怎么样？答案是：代码仍然正确执行。C++ 虽然不允许定义长度为 0 的数组变量，但明确指出，调用 new 动态创建长度为 0 的数组是合法的：

```
char arr[0]; // error: cannot define zero-length array
char *cp = new char[0]; // ok: but cp can't be dereferenced
```

用 new 动态创建长度为 0 的数组时，new 返回有效的非零指针。该指针与 new 返回的其他指针不同，不能进行解引用操作，因为它毕竟没有指向任何元素。而允许的操作包括：比较运算，因此该指针能在循环中使用；在该指针上加（减）0；或者减去本身，得 0 值。

在上述例题中，如果 get_size 返回 0，则仍然可以成功调用 new，但是 p 并没有指向任何对象，数组是空的。因为 n 为 0，所以 for 循环实际比较的是 p 和 q，而 q 是用 p 初始化的，两者具有相等的值，因此 for 循环条件不成立，循环体一次都没有执行。

5. 动态空间的释放

动态分配的内存最后必须进行释放，否则，内存最终将会逐渐耗尽。如果不再需要使用动态创建的数组，程序员必须显式地将其占用的存储空间返还给程序的自由存储区。C++ 语言为指针提供 delete [] 表达式释放指针所指向的数组空间：

```
delete [] pia;
```

该语句回收了 pia 所指向的数组，把相应的内存返还给自由存储区。在关键字 delete 和指针之间的空方括号对是必不可少的：它告诉编译器该指针指向的是自由存储区中的数组，而并非单个对象。



如果遗漏了空方括号对，这是一个编译器无法发现的错误，将导致程序在运行时出错。

理论上，回收数组时缺少空方括号对，至少会导致运行时少释放了内存空间，从而产生内存泄漏 (memory leak)。对于某些系统和/或元素类型，有可能会带来更严重的运行时错误。因此，在释放动态数组时千万别忘了方括号对。

137

C 风格字符串与 C++ 的标准库类型 string 的比较

以下两段程序反映了使用 C 风格字符串与 C++ 的标准库类型 string 的不同之处。使用 string 类型的版本更短、更容易理解，而且出错的可能性更小：

```
// C-style character string implementation
const char *pc = "a very long literal string";
const size_t len = strlen(pc + 1); // space to allocate
// performance test on string allocation and copy
for (size_t ix = 0; ix != 1000000; ++ix) {
    char *pc2 = new char[len + 1]; // allocate the space
    strcpy(pc2, pc); // do the copy
    if (strcmp(pc2, pc)) // use the new string
        ; // do nothing
    delete [] pc2; // free the memory
}

// string implementation
string str("a very long literal string");
// performance test on string allocation and copy
for (int ix = 0; ix != 1000000; ++ix) {
    string str2 = str; // do the copy, automatically allocated
    if (str != str2) // use the new string
        ; // do nothing
} // str2 is automatically freed
```

这些程序将在 4.3.1 节的习题中做进一步探讨。

6. 动态数组的使用

通常是因为在编译时无法知道数组的维数，所以才需要动态创建该数组。例如，在程序执行过程中，常常使用 char* 指针指向多个 C 风格字符串，于是必须根据每个字符串的长度实时地动态分配存储空间。采用这种技术要比建立固定大小的数组安全。如果程序员能够准确计算出运行时需要数组的长度，就不必再担心因数组变量具有固定的长度而造成的溢出问题。

假设有以下 C 风格字符串：

```
const char *noerr = "success";
// ...
const char *err189 = "Error: a function declaration must "
                    "specify a function return type!";
```

138

我们想在运行时把这两个字符串中的一个复制给新的字符数组，于是可以用以下程序在运行时计算维数：

```
const char *errorTxt;
if (errorFound)
    errorTxt = err189;
else
    errorTxt = noerr;
// remember the 1 for the terminating null
int dimension = strlen(errorTxt) + 1;
char *errMsg = new char[dimension];
// copy the text for the error into errMsg
strncpy (errMsg, errorTxt, dimension);
```

别忘记标准库函数 `strlen` 返回的是字符串的长度，并不包括字符串结束符，在获得的字符串长度上必须加 1 以便在动态分配时预留结束符的存储空间。

习题

习题 4.27 假设有下面的 `new` 表达式，请问如何释放 `pa`？

```
int *pa = new int[10];
```

习题 4.28 编写程序由从标准输入设备读入的元素数据建立一个 `int` 型 `vector` 对象，然后动态创建一个与该 `vector` 对象大小一致的数组，把 `vector` 对象的所有元素复制给新数组。

习题 4.29 对本小节第 5 条框中的两段程序：

(a) 解释这两个程序实现什么功能？

(b) 平均来说，使用 `string` 类型的程序执行速度要比用 C 风格字符串的快很多，在我们用了五年的 PC 机上其平均执行速度分别是：

```
user      0.47    # string class
user      2.55    # C-style character string
```

你预计的也一样吗？请说明原因。

习题 4.30 编写程序连接两个 C 风格字符串字面值，把结果存储在一个 C 风格字符串中。然后再编写程序连接两个 `string` 类型字符串，这两个 `string` 类型字符串与前面的 C 风格字符串字面值具有相同的内容。

4.3.2 新旧代码的兼容

许多 C++ 程序在有标准类之前就已经存在了，因此既没有使用标准库类型 `string` 也没有使用 `vector`。而且，许多 C++ 程序为了兼容现存的 C 程序，也不能使用 C++ 标准库。因此，现代的 C++ 程序经常必须兼容使用数组和/或 C 风格字符串的代码，标准库提供了使兼容界面更容易

139

管理的手段。

1. 混合使用标准库类 **string** 和 C 风格字符串

正如 3.2.1 节中显示的, 可用字符串字面值初始化 **string** 类对象:

```
string st3("Hello World"); // st3 holds Hello World
```

通常, 由于 C 风格字符串与字符串字面值具有相同的数据类型, 而且都是以空字符 **null** 结束, 因此可以把 C 风格字符串用在任何可以使用字符串字面值的地方:

- 可以使用 C 风格字符串对 **string** 对象进行初始化或赋值。
- **string** 类型的加法操作需要两个操作数, 可以使用 C 风格字符串作为其中的一个操作数, 也允许将 C 风格字符串用作复合赋值操作的右操作数。

反之则不成立: 在要求 C 风格字符串的地方不可直接使用标准库 **string** 类型对象。例如, 无法使用 **string** 对象初始化字符指针:

```
char *str = st2; // compile-time type error
```

但是, **string** 类提供了一个名为 **c_str** 的成员函数, 以实现我们的要求:

```
char *str = st2.c_str(); // almost ok, but not quite
```

c_str 函数返回 C 风格字符串, 其字面意思是: “返回 C 风格字符串的表示方法”, 即返回指向字符数组首地址的指针, 该数组存放了与 **string** 对象相同的内容, 并且以结束符 **null** 结束。

因为 **c_str** 返回的指针指向 **const char** 类型的数组, 所以上述初始化失败, 这样做是为了避免修改该数组。正确的初始化应为:

```
const char *str = st2.c_str(); // ok
```



c_str 返回的数组并不保证一定是有效的, 接下来对 **st2** 的操作有可能会改变 **st2** 的值, 使刚才返回的数组失效。如果程序需要持续访问该数据, 则应该复制 **c_str** 函数返回的数组。

2. 使用数组初始化 **vector** 对象

4.1.1 节提到不能用一个数组直接初始化另一数组, 程序员只能创建新数组, 然后显式地把源数组的元素逐个复制给新数组。这反映 C++ 允许使用数组初始化 **vector** 对象, 尽管这种初始化形式起初看起来有点陌生。使用数组初始化 **vector** 对象, 必须指出用于初始化式的第一个元素以及数组最后一个元素的下一位置的地址:

```
const size_t arr_size = 6;
int int_arr[arr_size] = {0, 1, 2, 3, 4, 5};
// ivec has 6 elements: each a copy of the corresponding element in int_arr
vector<int> ivec(int_arr, int_arr + arr_size);
```

传递给 **ivec** 的两个指针标出了 **vector** 初值的范围。第二个指针指向被复制的最后一个元素之后的地址空间。被标出的元素范围可以是数组的子集:

```
// copies 3 elements: int_arr[1], int_arr[2], int_arr[3]
vector<int> ivec(int_arr + 1, int_arr + 4);
```

这个初始化创建了含有三个元素的 `ivec`，三个元素的值分别是 `int_arr[1]` 到 `int_arr[3]` 的副本。

习题

- 习题 4.31** 编写程序从标准输入设备读入字符串，并把该串存放在字符数组中。描述你的程序如何处理可变长的输入。提供比你分配的数组长度长的字符串数据测试你的程序。
- 习题 4.32** 编写程序用 `int` 型数组初始化 `vector` 对象。
- 习题 4.33** 编写程序把 `int` 型 `vector` 复制给 `int` 型数组。
- 习题 4.34** 编写程序读入一组 `string` 类型的数据，并将它们存储在 `vector` 中。接着，把该 `vector` 对象复制给一个字符指针数组。为 `vector` 中的每个元素创建一个新的字符数组，并把该 `vector` 元素的数据复制到相应的字符数组中，最后把指向该数组的指针插入字符指针数组。
- 习题 4.35** 输出习题 4.34 中建立的 `vector` 对象和数组的内容。输出数组后，记得释放字符数组。

4.4 多维数组



严格地说，C++ 中没有多维数组，通常所指的多维数组其实就是数组的数组：

```
// array of size 3, each element is an array of ints of size 4
int ia[3][4];
```

在使用多维数组时，记住这一点有利于理解其应用。

如果数组的元素又是数组，则称为二维数组，其每一维对应一个下标：

141

```
ia[2][3] // fetches last element from the array in the last row
```

第一维通常称为行 (row)，第二维则称为列 (column)。C++ 中并未限制可用的下标个数，也就是说，我们可以定义元素是数组（其元素又是数组，如此类推）的数组。

1. 多维数组的初始化

和处理一维数组一样，程序员可以使用由花括号括起来的初始化式列表来初始化多维数组的元素。对于多维数组的每一行，可以再用花括号指定其元素的初始化式：

```
int ia[3][4] = {           /* 3 elements, each element is an array of size 4 */
    {0, 1, 2, 3},          /* initializers for row indexed by 0 */
    {4, 5, 6, 7},          /* initializers for row indexed by 1 */
    {8, 9, 10, 11}         /* initializers for row indexed by 2 */
};
```

其中用来标志每一行的内嵌的花括号是可选的。下面的初始化尽管有点不清楚，但与前面的声明完全等价：

```
// equivalent initialization without the optional nested braces for each row
int ia[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

与一维数组一样，有些元素将不使用初始化列表提供的初始化式进行初始化。下面的声明只初始化了每行的第一个元素：

```
// explicitly initialize only element 0 in each row
int ia[3][4] = {{ 0 }, { 4 }, { 8 } };
```

其余元素根据其元素类型用 4.1.1 节描述的规则初始化。

如果省略内嵌的花括号，结果会完全不同：

```
// explicitly initialize row 0
int ia[3][4] = {0, 3, 6, 9};
```

该声明初始化了第一行的元素，其余元素都被初始化为 0。

2. 多维数组的下标引用

为了对多维数组进行索引，每一维都需要一个下标。例如，下面的嵌套 for 循环初始化了一个二维数组：

```
const size_t rowSize = 3;
const size_t colSize = 4;
int ia [rowSize][colSize]; // 12 uninitialized elements
// for each row
for (size_t i = 0; i != rowSize; ++i)
    // for each column within the row
    for (size_t j = 0; j != colSize; ++j)
        // initialize to its positional index
        ia[i][j] = i * colSize + j;
```

142

当需要访问数组中的特定元素时，必须提供其行下标和列下标。行下标指出需要哪个内部数组，列下标则选取该内部数组的指定元素。了解多维数组下标引用策略有助于正确计算其下标值，以及理解多维数组如何初始化。

如果表达式只提供了一个下标，则结果获取的元素是该行下标索引的内层数组。如 `ia[2]` 将获得 `ia` 数组的最后一行，即这一行的内层数组本身，而并非该数组中的任何元素。

指针和 multidimensional array

与普通数组一样，使用多维数组名时，实际上将其自动转换为指向该数组第一个元素的指针。



定义指向多维数组的指针时，千万别忘了该指针所指向的多维数组其实是数组的数组。

因为多维数组其实就是数组的数组，所以由多维数组转换而成的指针类型应是指向第一个内层数组的指针。尽管这个概念非常明了，但声明这种指针的语法还是不容易理解：

```
int ia[3][4]; // array of size 3, each element is an array of ints of size 4
int (*ip)[4] = ia; // ip points to an array of 4 ints
ip = &ia[2]; // ia[2] is an array of 4 ints
```

定义指向数组的指针与如何定义数组本身类似：首先声明元素类型，后接（数组）变量名字和维数。窍门在于（数组）变量的名字其实是指针，因此需在标识符前加上*。如果从内向外阅

读 `ip` 的声明，则可理解为：`*ip` 是 `int[4]` 类型——即 `ip` 是一个指向含有 4 个元素的数组的指针。



在下面的声明中，圆括号是必不可少的：

```
int *ip[4];           // array of pointers to int
int (*ip)[4];        // pointer to an array of 4 ints
```

用 **typedef** 简化指向多维数组的指针

`typedef` 类型定义（2.6 节）可使指向多维数组元素的指针更容易读、写和理解。以下程序用 `typedef` 为 `ia` 的元素类型定义新的类型名：

```
typedef int int_array[4];
int_array *ip = ia;
```

可使用 `typedef` 类型输出 `ia` 的元素：

```
for (int_array *p = ia; p != ia + 3; ++p)
    for (int *q = *p; q != *p + 4; ++q)
        cout << *q << endl;
```

143

外层的 `for` 循环首先初始化 `p` 指向 `ia` 的第一个内部数组，然后一直循环到 `ia` 的三行数据都处理完为止。`++p` 使 `p` 加 1，等效于移动指针使其指向 `ia` 的下一行（例如：下一个元素）。

内层的 `for` 循环实际上处理的是存储在内部数组中的 `int` 型元素值。首先让 `q` 指向 `p` 所指向的数组的第一个元素。对 `p` 进行解引用获得一个有 4 个 `int` 型元素的数组，通常，使用这个数组时，系统会自动将它转换为指向该数组第一个元素的指针。在本例中，第一个元素是 `int` 型数据，`q` 指向这个整数。系统执行内层的 `for` 循环直到处理完当前 `p` 指向的内部数组中所有的元素为止。当 `q` 指针刚达到该内部数组的超出末端位置时，再次对 `p` 进行解引用以获得指向下一个内部数组第一个元素的指针。在 `p` 指向的地址上加 4 使得系统可循环处理每一个内部数组的 4 个元素。

习题

144

习题 4.36 重写程序输出 `ia` 数组的内容，要求在外层循环中不能使用 `typedef` 定义的类型。

小结

本章介绍了数组和指针。数组和指针所提供的功能类似于标准库的 `vector` 类与 `string` 类和相关的迭代器。我们可以把 `vector` 类型理解为更灵活、更容易管理的数组，同样，`string` 是 C 风格字符串的改进类型，而 C 风格字符串是以空字符结束的字符数组。

迭代器和指针都能用于间接地访问所指向的对象。`vector` 类型所包含的元素通过迭代器来操纵，类似地，指针则用于访问数组元素。尽管道理都很简单，但在实际应用中，指针的难用是出了名的。

某些低级任务必须使用指针和数组，但由于使用指针和数组容易出错而且难以调试，应尽量避免使用。一般而言，应该优先使用标准库抽象类而少用语言内置的低级数组和指针。尤其是应该使用 `string` 类型取代 C 风格以空字符结束的字符数组。现代 C++ 程序不应使用 C 风格字符串。

术语

C-style string (C 风格字符串) C 程序把指向以空字符结束的字符数组的指针视为字符串。在 C++ 中，字符串字面值就是 C 风格字符串。C 标准库定义了一系列处理这种字符串的库函数，C++ 中将这此标准库函数放在 `cstring` 头文件中。由于 C 风格字符串本质上容易出错，C++ 程序应该优先使用 C++ 标准库类 `string` 而少用 C 风格字符串。网络程序中大量的安全漏洞都源于与使用 C 风格字符串和数组相关的缺陷。

compiler extension (编译器扩展) 特定编译器为语言添加的特性。依赖于编译器扩展的程序很难移植到其他的编译器。

compound type (复合类型) 使用其他类型定义的类型。数组、指针和引用都是复合类型。

const void* 可以指向任意 `const` 类型的指针类型，参见 `void *`。

delete expression (delete 表达式) `delete` 表达式用于释放由 `new` 动态分配的内存：

```
delete [] p;
```

在此表达式中，`p` 必须是指向动态创建的数组中第一个元素的指针，其中方括号必不可少：它告诉编译器该指针指向数组，而非单个对象。C++ 程序使用 `delete` 取代 C 语言的标准库函数 `free`。

dimension (维数) 数组大小。

dynamically allocated (动态分配的) 在程序自由存储区中建立的对象。该对象一经创建就一直存在，直到显式释放为止。

free store (自由存储区) 程序用来存储动态创建对象的内存区域。

heap (堆) 自由存储区的同义词。

new expression (new 表达式) 用于分配动态内存的表达式。下面的语句分配了一个有 `n` 个元素的数组：

```
new type[n];
```

该数组存放 `type` 类型的元素。`new` 返回指向该数组第一个元素的指针。C++ 程序使用 `new` 取代 C 语言的标准库函数 `malloc`。

pointer (指针) 存放对象地址的对象。

pointer arithmetic (指针算术操作) 可用于指针的算术操作。允许在指针上做加上或减去整型值的操作，以获得当前指针之前或之后若干个元素处的地址。两个指针可做减法操作，得到它们之间的差值。只有当指针指向同一个数组或其超出末端的位置时，指针的算术操作才有意义。

precedence (优先级) 在复杂的表达式中，优先级确定了操作数分组的次序。

ptrdiff_t 在 `cstdint` 头文件中定义的与机器相关的有符号整型，该类型具有足够的大小存储两个指针的差值，这两个指针指向同一个可能的最大数组。

size_t 在 `cstdint` 头文件中定义的与机器相关的无符号整型，它具有足够的大小存储一个可能的最大数组。

*** operator (* 操作符)** 对指针进行解引用操作获得该指针所指向的对象。解引用操作符返回左值，因此可为其结果赋值，等效于为该指针所指向的特定对象赋新值。

++ operator (++ 操作符) 用于指针时，自增操作符给指针“加 1”，移动指针使其指向数组

的下一个元素。

[] operator ([] 操作符) 下标操作符接受两个操作数：一个是指向数组元素的指针，一个是下标 n 。该操作返回偏离指针当前指向 n 个位置的元素值。数组下标从 0 开始计数——数组第一个元素的下标为 0，最后一个元素的下标是数组长度减 1。下标操作返回左值，可用做赋值操作的左操作数，等效于为该下标引用

的元素赋新值。

& operator (& 操作符) 取地址操作符需要一个操作数，其唯一的操作数必须是左值对象，该操作返回操作数对象在内存中的存储地址。**void*** 可以指向任何非 **const** 对象的指针类型。**void*** 指针只提供有限的几种操作：可用作函数形参类型或返回类型，也可与其他指针做比较操作，但是不能进行解引用操作。

145
146

第 5 章

表 达 式

目录

5.1 算术操作符.....	129
5.2 关系操作符和逻辑操作符.....	131
5.3 位操作符.....	134
5.4 赋值操作符.....	137
5.5 自增和自减操作符.....	140
5.6 箭头操作符.....	142
5.7 条件操作符.....	143
5.8 sizeof 操作符.....	144
5.9 逗号操作符.....	145
5.10 复合表达式的求值.....	145
5.11 new 和 delete 表达式.....	150
5.12 类型转换.....	154
小结.....	161
术语.....	162

C++提供了丰富的操作符，并定义操作数为内置类型时，这些操作符的含义。除此之外，C++还支持操作符重载，允许程序员自定义用于类类型时操作符的含义。标准库正是使用这种功能定义用于库类型的操作符。

本章重点介绍 C++语言定义的操作符，它们使用内置类型的操作数；本章还会介绍一些标准库定义的操作符。第 14 章将学习如何定义自己的重载操作符。

表达式由一个或多个操作数 (operand) 通过操作符 (operator) 组合而成。最简单的表达式 (expression) 仅包含一个字面值常量或变量。较复杂的表达式则由操作符以及一个或多个操作数构成。

每个表达式都会产生一个结果 (result)。如果表达式中没有操作符, 则其结果就是操作数本身 (例如, 字面值常量或变量) 的值。当一个对象用在需要使用其值的地方, 则计算该对象的值。例如, 假设 `ival` 是一个 `int` 型对象:

```
if (ival)           // evaluate ival as a condition
    // ....
```

上述语句将 `ival` 作为 `if` 语句的条件表达式。当 `ival` 为非零值时, `if` 条件成立; 否则条件不成立。

对于含有操作符的表达式, 它的值通过对操作数做指定操作获得。除了特殊用法外, 表达式的结果是右值 (2.3.1 节), 可以读取该结果值, 但是不允许对它进行赋值。



操作符的含义——该操作符执行什么操作以及操作结果的类型——取决于操作数的类型。

除非已知道操作数的类型, 否则无法确定一个特定表达式的含义。下面的表达式

```
i + j
```

既可能是整数的加法操作、字符串的串接或者浮点数的加法操作, 也完全可能是其他的操作。如何计算该表达式的值, 完全取决于 `i` 和 `j` 的数据类型。

C++ 提供了一元操作符 (unary operator) 和二元操作符 (binary operator) 两种操作符。作用在一个操作数上的操作符称为一元操作符, 如取地址操作符 (&) 和解引用操作符 (*); 而二元操作符则作用于两个操作数上, 如加法操作符 (+) 和减法操作符 (-)。除此之外, C++ 还提供了一个使用三个操作数的三元操作符 (ternary operator), 我们将在 5.7 节介绍它。

有些符号 (symbol) 既可表示一元操作也可表示二元操作。例如, 符号 * 既可以作为 (一元) 解引用操作符, 也可以作为 (二元) 乘法操作符, 这两种用法相互独立、各不相关, 如果将其视为两个不同的符号可能会更容易理解些。对于这类操作符, 需要根据该符号所处的上下文来确定它代表一元操作还是二元操作。

操作符对其操作数的类型有要求, 如果操作符应用于内置或复合类型的操作数, 则由 C++ 语言定义其类型要求。例如, 用于内置类型对象的解引用操作符要求其操作数必须是指针类型, 对任何其他内置类型或复合类型对象进行解引用将导致错误的产生。

对于操作数为内置或复合类型的二元操作符, 通常要求它的两个操作数具有相同的数据类型, 或者其类型可以转换为同一种数据类型。关于类型转换, 我们将在 5.12 节学习。尽管规则可能比较复杂, 但大部分的类型转换都可按预期的方式进行。例如, 整型可转换为浮点类型, 反之亦然, 但不能将指针类型转换为浮点类型。

148

要理解由多个操作符组成的表达式, 必须先理解操作符的优先级 (precedence)、结合性 (associativity) 和操作数的求值顺序 (order of evaluation)。例如, 表达式

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

使用了加法、乘法和除法操作。该表达式的值取决于操作数与操作符如何结合。例如，乘法操作符*的操作数可以是 10 和 20，也可以是 10 和 20/2，或者 15 和 20、15 和 20/2。结合性和优先级规则规定了操作数与操作符的结合方式。在 C++语言中，该表达式的值应是 105，10 和 20 先做乘法操作，然后其结果除以 2，再加 5 即为最后结果。

求解表达式时，仅了解操作数和操作符如何结合是不足够的，还必须清楚操作符上每一个操作数的求值顺序。每个操作符都控制了其假定的求值顺序，即，我们是否可以假定左操作数总是先于右操作数求值。大部分的操作符无法保证某种特定的求值次序，我们将于 5.10 节讨论这个问题。

5.1 算术操作符

除非特别说明，表 5-1 所示操作符可用于任意算术类型（2.1 节）或者任何可转换为算术类型的数据类型。

表 5-1 按优先级来对操作符进行分组——一元操作符优先级最高，其次是乘、除操作，接着是二元的加、减法操作。高优先级的操作符要比低优先级的结合得更紧密。这些算术操作符都是左结合，这就意味着当操作符的优先级相同时，这些操作符从左向右依次与操作数结合。

149

表 5-1 算术操作符		
操作符	功 能	用 法
+	一元正号	+ expr
-	一元负号	- expr
*	乘法	expr * expr
/	除法	expr / expr
%	求余	expr % expr
+	加法	expr + expr
-	减法	expr - expr

注：expr 为表达式。

对于前述表达式

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

考虑优先级与结合性，可知该表达式先做乘法（*）操作，其操作数为 10 和 20，然后以该操作的结果和 2 为操作数做除法（/）操作，其结果最后与操作数 5 做加法（+）操作。

一元负号操作符具有直观的含义，它对其操作数取负：

```
int i = 1024;
int k = -i; // negates the value of its operand
```

一元正号操作符则返回操作数本身，对操作数不作任何修改。

警告：溢出和其他算术异常

某些算术表达式的求解结果未定义，其中一部分由数学特性引起，例如除零操作；其他则归咎于计算机特性，如溢出：计算出的数值超出了其类型的表示范围。

考虑某台机器，其 short 类型为 16 位，能表示的最大值是 32767。假设 short 类型只有 16 位，下面的复合赋值操作将会溢出：

```
// max value if shorts are 8 bits
short short_value = 32767;
short ival = 1;
// this calculation overflows
short_value += ival;
cout << "short_value: " << short_value << endl;
```

表示 32768 这个有符号数需 17 位的存储空间，但是这里仅有 16 位，于是导致溢出现象的发生，此时，许多系统都不会给出编译时或运行时的警告。对于不同的机器，上述例子的 short_value 变量真正获得的值不尽相同。在我们的系统上执行该程序后将得到：

```
short_value: -32768
```

其值“截断 (wrapped around)”，将符号位的值由 0 设为 1，于是结果变为负数。因为算术类型具有有限的长度，因此计算后溢出的现象常常发生。遵循 2.2 节建议框中给出的建议将有助于避免此类问题。

二元+、-操作符也可用于指针值，对指针使用这些操作符的用法将在 4.2.4 节介绍。

算术操作符 +、-、* 和 / 具有直观的含义：加法、减法、乘法和除法。对两个整数做除法，结果仍为整数，如果它的商包含小数部分，则小数部分会被截除：

```
int ival1 = 21/6; // integral result obtained by truncating the remainder
int ival2 = 21/7; // no remainder, result is an integral value
```

150 ival1 和 ival2 均被初始化为 3。

操作符 % 称为“求余 (remainder)”或“求模 (modulus)”操作符，用于计算左操作数除以右操作数的余数。该操作符的操作数只能为整型，包括 bool、char、short、int 和 long 类型，以及对应的 unsigned 类型：

```
int ival = 42;
double dval = 3.14;
ival % 12; // ok: returns 6
ival % dval; // error: floating point operand
```

如果两个操作数为正，除法 (/) 和求模 (%) 操作的结果也是正数（或零）；如果两个操作数都是负数，除法操作的结果为正数（或零），而求模操作的结果则为负数（或零）；如果只有一个操作数为负数，这两种操作的结果取决于机器；求模结果的符号也取决于机器，而除法操作的值则是负数（或零）：

```
21 % 6; // ok: result is 3
21 % 7; // ok: result is 0
-21 % -8; // ok: result is -5
```

```
21 % -5; // machine-dependent: result is 1 or -4
21 / 6; // ok: result is 3
21 / 7; // ok: result is 3
-21 / -8; // ok: result is 2
21 / -5; // machine-dependent: result -4 or -5
```

当只有一个操作数为负数时，求模操作结果值的符号可依据分子（被除数）或分母（除数）的符号而定。如果求模的结果随分子的符号，则除出来的值向零一侧取整；如果求模与分母的符号匹配，则除出来的值向负无穷一侧取整。

习题

- 习题 5.1 在下列表达式中，加入适当的圆括号以标明其计算顺序。编译该表达式并输出其值，从而检查你的回答是否正确。
- 12 / 3 * 4 + 5 * 15 + 24 % 4 / 2
- 习题 5.2 计算下列表达式的值，并指出哪些结果值依赖于机器？
- 30 * 3 + 21 / 5
-30 + 3 * 21 / 5
30 / 3 * 21 % 5
-30 / 3 * 21 % 4
- 习题 5.3 编写一个表达式判断一个 int 型数值是偶数还是奇数。
- 习题 5.4 定义术语“溢出”的含义，并给出导致溢出的三个表达式。

5.2 关系操作符和逻辑操作符

关系操作符和逻辑操作符（表 5-2）使用算术或指针类型的操作数，并返回 bool 类型的值。

表 5-2 关系操作符和逻辑操作符		
下列操作符都产生 bool 值		
操作符	功 能	用 法
!	逻辑非	!expr
<	小于	expr < expr
<=	小于等于	expr <= expr
>	大于	expr > expr
>=	大于等于	expr >= expr
==	相等	expr == expr
!=	不等	expr != expr
&&	逻辑与	expr && expr
	逻辑或	expr expr

1. 逻辑与、逻辑或操作符

逻辑操作符将其操作数视为条件表达式（1.4.1 节）：首先对操作数求值；若结果为 0，则条件为假（false），否则为真（true）。仅当逻辑与（&&）操作符的两个操作数都为 true，其结果才得 true。对于逻辑或（||）操作符，只要两个操作数之一为 true，它的值就为 true。给

定以下形式:

```
expr1 && expr2 // logical AND
expr1 || expr2 // logical OR
```

仅当由 *expr1* 不能确定表达式的值时, 才会求解 *expr2*。也就是说, 当且仅当下列情况出现时, 必须确保 *expr2* 是可以计算的:

- 在逻辑与表达式中, *expr1* 的计算结果为 true。如果 *expr1* 的值为 false, 则无论 *expr2* 的值是什么, 逻辑与表达式的值都为 false。当 *expr1* 的值为 true 时, 只有 *expr2* 的值也是 true, 逻辑与表达式的值才为 true。
- 在逻辑或表达式中, *expr1* 的计算结果为 false。如果 *expr1* 的值为 false, 则逻辑或表达式的值取决于 *expr2* 的值是否为 true。



逻辑与和逻辑或操作符总是先计算其左操作数, 然后再计算其右操作数。只有在仅靠左操作数的值无法确定该逻辑表达式的结果时, 才会求解其右操作数。我们常常称这种求值策略为“短路求值 (short-circuit evaluation)”。

152

对于逻辑与操作符, 一个很有价值的用法是: 如果某边界条件使 *expr2* 的计算变得危险, 则应在该条件出现之前, 先让 *expr1* 的计算结果为 false¹。例如, 编写程序使用一个 string 类型的对象存储一个句子, 然后将该句子的第一个单词的各字符全部变成大写, 可如下实现:

```
string s("Expressions in C++ are composed...");
string::iterator it = s.begin();
// convert first word in s to uppercase
while (it != s.end() && !isspace(*it)) {
    *it = toupper(*it); // toupper covered in section 3.2.4
    ++it;
}
```

在这个例子中, while 循环判断了两个条件。首先检查 *it* 是否已经到达 string 类型对象的结尾, 如果不是, 则 *it* 指向 *s* 中的一个字符。只有当该检验条件成立时, 系统才会计算逻辑与操作符的右操作数, 即在保证 *it* 确实指向一个真正的字符之后, 才检查该字符是否为空格。如果遇到空格, 或者 *s* 中没有空格而已经到达 *s* 的结尾时, 循环结束。

2. 逻辑非操作符

逻辑非操作符 (!) 将其操作数视为条件表达式, 产生与其操作数值相反的条件值。如果其操作数为非零值, 则做!操作后的结果为 false。例如, 可如下在 vector 类型对象的 empty 成员函数上使用逻辑非操作符, 根据函数返回值判断该对象是否为空:

```
// assign value of first element in vec to x if there is one
int x = 0;
if (!vec.empty())
    x = *vec.begin();
```

如果调用 empty 函数返回 false, 则子表达式

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

1. 因为当 *expr1* 为 false 时, 不再求解 *expr2*。——译者注

的值为 true。

3. 不应该串接使用关系操作符

关系操作符 (<、<=、>、>=) 具有左结合特性。事实上，由于关系操作符返回 bool 类型的结果，因此很少使用其左结合特性。如果把多个关系操作符串接起来使用，结果往往出乎预料：

```
// oops! this condition does not determine if the 3 values are unequal
if (i < j < k) { /* ... */ }
```

这种写法只要 k 大于 1，上述表达式的值就为 true。这是因为第二个小于操作符的左操作数是第一个小于操作符的结果：true 或 false。也就是，该条件将 k 与整数 0 或 1 做比较。为了实现我们想要的条件检验，应重写上述表达式如下：

```
if (i < j && j < k) { /* ... */ }
```

153

4. 相等测试与 bool 字面值

正如 5.12.2 节将介绍的，bool 类型可转换为任何算术类型——bool 值 false 用 0 表示，而 true 则为 1。



由于 true 转换为 1，因此要检测某值是否与 bool 字面值 true 相等，其等效判断条件通常很难正确编写：

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

val 本身是 bool 类型，或者 val 具有可转换为 bool 类型的数据类型。如果 val 是 bool 类型，则该判断条件等效于：

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

这样的代码更短而且更直接（尽管对初学者来说，这样的缩写可能会令人费解）。

更重要的是，如果 val 不是 bool 值，val 和 true 的比较等效于：

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

这与下面的条件判断完全不同：

```
// condition succeeds if val is any nonzero value
if (val) { /* ... */ }
```

此时，只要 val 为任意非零值，条件判断都得 true。如果显式地书写条件比较，则只有当 val 等于指定的 1 值时，条件才成立。

习题

习题 5.5 解释逻辑与操作符、逻辑或操作符以及相等操作符的操作数在什么时候计算。

习题 5.6 解释下列 while 循环条件的行为：

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

习题 5.7 编写 while 循环条件从标准输入设备读入整型(int)数据，当读入值为 42 时循环结束。

习题 5.8 编写表达式判断四个值 a、b、c 和 d 是否满足 a 大于 b、b 大于 c 而且 c 大于 d 的条件。

5.3 位操作符

位操作符（表 5-3）使用整型的操作数。位操作符将其整型操作数视为二进制位的集合，为每一位提供检验和设置的功能。另外，这类操作符还可用于 `bitset` 类型（3.5 节）的操作数，该类型具有这里所描述的整型操作数的行为。

154

表 5-3 位操作符		
操作符	功 能	用 法
<code>~</code>	位求反	<code>~expr</code>
<code><<</code>	左移	<code>expr1 << expr2</code>
<code>>></code>	右移	<code>expr1 >> expr2</code>
<code>&</code>	位与	<code>expr1 & expr2</code>
<code>^</code>	位异或	<code>expr1 ^ expr2</code>
<code> </code>	位或	<code>expr1 expr2</code>

位操作符操纵的整数的类型可以是有符号的也可以是无符号的。如果操作数为负数，则位操作符如何处理其操作数的符号位依赖于机器。于是它们的应用可能不同：在一个应用环境中实现的程序可能无法用于另一应用环境。

 **最佳实践** 对于位操作符，由于系统不能确保如何处理其操作数的符号位，所以强烈建议使用 `unsigned` 整型操作数。

在下面的例子中，假设 `unsigned char` 类型有 8 位。
位求反操作符（`~`）的功能类似于 `bitset` 的 `flip` 操作（3.5.2 节）：将操作数的每一个二进制位取反：将 1 设置为 0、0 设置为 1，生成一个新值：

```
unsigned char bits = 0227;      1 0 0 1 0 1 1 1
bits = ~bits;                  0 1 1 0 1 0 0 0
```

`>>`和`<<`操作符提供移位操作，其右操作数标志要移动的位数。这两种操作符将其左操作数的各个位向左（`<<`）或向右（`>>`）移动若干个位（移动的位数由其右操作数指定），从而产生新的值，并丢弃移出去的位。

```
unsigned char bits = 1;         1 0 0 1 1 0 1 1
bits << 1; // left shift        0 0 1 1 0 1 1 0
bits << 2; // left shift        0 1 1 0 1 1 0 0
bits >> 3; // right shift       0 0 0 1 0 0 1 1
```

左移操作符（`<<`）在右边插入 0 以补充空位。对于右移操作符（`>>`），如果其操作数是无符号数，则从左边开始插入 0；如果操作数是有符号数，则插入符号位的副本或者 0 值，如何选择需依据具体的实现而定。移位操作的右操作数不可以是负数，而且必须是严格小于左操作数位数的值。否则，操作的效果未定义。

155

位与操作 (&) 需要两个整型操作数，在每个位的位置，如果两个操作数对应的位都为 1，则操作结果中该位为 1，否则为 0。



常犯的错误是把位与操作 (&) 和逻辑与操作 (&&) (5.2 节) 混淆了。同样地，位或操作 (|) 和逻辑或操作 (||) 也很容易搞混。

下面我们用图解的方法说明两个 unsigned char 类型值的位与操作，这两个操作数均用八进制字面常量初始化：

```
unsigned char b1 = 0145;      01100101
unsigned char b2 = 0257;      10101111
unsigned char result = b1 & b2; 00100101
```

位异或（互斥或，exclusive or）操作符 (^) 也需要两个整型操作数。在每个位的位置，如果两个操作数对应的位只有一个（不是两个）为 1，则操作结果中该位为 1，否则为 0。

```
result = b1 ^ b2;      11001010
```

位或（包含或，inclusive or）操作符 (|) 需要两个整型操作数。在每个位的位置，如果两个操作数对应的位有一个或者两个都为 1，则操作结果中该位为 1，否则为 0。

```
result = b1 | b2;      11101111
```

5.3.1 bitset 对象或整型值的使用

bitset 类比整型值上的低级位操作更容易使用。观察下面简单的例子，了解如何使用 bitset 类型或者位操作来解决问题。假设某老师带了一个班，班中有 30 个学生，每个星期在班上做一次测验，只有及格和不及格两种测验成绩，对每个学生用一个二进制位来记录一次测试及格或不及格，以方便我们跟踪每次测验的结果，这样就可以用一个 bitset 对象或整数值来代表一次测验：

```
bitset<30> bitset_quiz1;      // bitset solution
unsigned long int_quiz1 = 0;   // simulated collection of bits
```

使用 bitset 类型时，可根据所需要的大小明确地定义 bitset_quiz1，它的每一个位都默认设置为 0 值。如果使用内置类型来存放测验成绩，则应将变量 int_quiz1 定义为 unsigned long 类型，这种数据类型在所有机器上都至少拥有 32 位的长度。最后，显式地初始化 int_quiz1 以保证该变量在使用前具有明确定义的值。

老师可以设置和检查每个位。例如，假设第 27 位所表示的学生及格了，则可以使用下面的语句适当地设置对应的位：

```
bitset_quiz1.set(27);        // indicate student number 27 passed
int_quiz1 |= 1UL<<27;        // indicate student number 27 passed
```

如果使用 bitset 实现，可直接传递要置位的位给 set 函数。而用 unsigned long 实现时，实

现的方法则比较复杂。设置指定位的方法是：将测验数据与一个整数做位或操作，该整数只有一个指定的位为 1。也就是说，我们需要一个只有第 27 位为 1 其他位都为 0 的无符号长整数 (unsigned long)，这样的整数可用左移操作符和整型常量 1 生成：

```
1UL << 27; // generate a value with only bit number 27 set
```

然后让这个整数与 int_quiz1 做位或操作，操作后，除了第 27 位外其他所有位的值都保持不变，而第 27 位则被设置为 1。这里，使用复合赋值操作 (1.4.1 节) 将位或操作的结果赋给 int_quiz1，该操作符 |= 操作的方法与 += 相同。于是，上述功能等效于下面更详细的形式：

```
// following assignment is equivalent to int_quiz1 |= 1UL << 27;
int_quiz1 = int_quiz1 | 1UL << 27;
```

如果老师重新复核测验成绩，发现第 27 个学生实际上在该次测验中不及格，这时老师应把第 27 位设置为 0：

```
bitset_quiz1.reset(27); // student number 27 failed
int_quiz1 &= ~(1UL<<27); // student number 27 failed
```

使用 bitset 的版本可直接实现该功能，只要复位 (reset) 指定的位即可。而对于另一种情况，则需通过反转左移操作后的结果来实现设置：此时，我们需要一个只有第 27 位为 0 而其他位都为 1 的整数。然后将这个整数与测验数据做位与操作，把指定的位设置为 0。位求反操作使得除了第 27 位外其他位都设置为 1，然后此值和 int_quiz1 做位与操作，保证了除第 27 位外所有的位都保持不变。

最后，可通过以下代码获知第 27 个学生是否及格：

```
bool status;
status = bitset_quiz1[27]; // how did student number 27 do?
status = int_quiz1 & (1UL<<27); // how did student number 27 do?
```

使用 bitset 的版本中，可直接读取其值判断他是否及格。使用 unsigned long 时，首先要把一个整数的第 27 位设置为 1，然后用该整数和 int_quiz1 做位与操作，如果 int_quiz1 的第 27 位为 1，则结果为非零值，否则，结果为零。

157

最佳
实践

一般而言，标准库提供的 bitset 操作更直接、更容易阅读和书写、正确使用的可能性更高。而且，bitset 对象的大小不受 unsigned 数的位数限制。通常来说，bitset 优于整型数据的低级直接位操作。

习题

习题 5.9 假设有下面两个定义

```
unsigned long ul1 = 3, ul2 = 7;
下列表达式的结果是什么？
```

- | | |
|----------------|----------------|
| (a) ul1 & ul2 | (c) ul1 ul2 |
| (b) ul1 && ul2 | (d) ul1 ul2 |

习题 5.10 重写 bitset 表达式：使用下标操作符对测验结果进行置位 (置 1) 和复位 (置 0)。

5.3.2 将移位操作符用于 IO

输入输出标准库 (IO library) 分别重载了位操作符>>和<<用于输入和输出。即使很多程序员从未直接使用过位操作符,但是相当多的程序都大量用到这些操作符在 IO 标准库中的重载版本。重载的操作符与该操作符的内置类型版本有相同的优先级和结合性。因此,即使程序员从不使用这些操作符的内置含义来实现移位操作,但是还是应该先了解这些操作符的优先级和结合性。

IO 操作符为左结合

像其他二元操作符一样,移位操作符也是左结合的。这类操作符从左向右地结合,正好说明了程序员为什么可以把多个输入或输出操作连接为单个语句:

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

执行为:

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

在这个语句中,操作数"hi"与第一个<<符号结合,其计算结果与第二个<<符号结合,第二个<<符号操作后,其结果再与第三个<<符号结合。

移位操作符具有中等优先级:其优先级比算术操作符低,但比关系操作符、赋值操作符和条件操作符优先级高。若 IO 表达式的操作数包含了比 IO 操作符优先级低的操作符,相关的优先级别将影响书写该表达式的方式。通常需使用圆括号强制先实现右结合:

```
cout << 42 + 10;      // ok, + has higher precedence, so the sum is printed
cout << (10 < 42);   // ok: parentheses force intended grouping; prints 1
cout << 10 < 42;      // error: attempt to compare cout to 42!
```

第二个 cout 语句解释为:

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

该表达式说“将 10 写到 cout,然后用此操作(也就是 cout)的结果与 42 做比较”。

5.4 赋值操作符

赋值操作符的左操作数必须是非 const 的左值。下面的赋值语句是不合法的:

```
int i, j, ival;
const int ci = i;    // ok: initialization not assignment
1024 = ival;         // error: literals are rvalues
i + j = ival;        // error: arithmetic expressions are rvalues
ci = ival;           // error: can't write to ci
```

数组名是不可修改的左值:因此数组不可用作赋值操作的目标。而下标和解引用操作符都返回左值,因此当将这两种操作用于非 const 数组时,其结果可作为赋值操作的左操作数:

```
int ia[10];
ia[0] = 0;           // ok: subscript is an lvalue
*ia = 0;             // ok: dereference also is an lvalue
```

赋值表达式的值是其左操作数的值，其结果的类型为左操作数的类型。

通常，赋值操作将其右操作数的值赋给左操作数。然而，当左、右操作数的类型不同时，该操作实现的类型转换可能会修改被赋的值。此时，存放在左、右操作数里的值并不相同：

```
ival = 0;           // result: type int value 0
ival = 3.14159;    // result: type int value 3
```

上述两个赋值语句都产生 `int` 类型的值，第一个语句中 `ival` 的值与右操作数的值相同；但是在第二个语句中，`ival` 的值则与右操作数的值不相同。

159

5.4.1 赋值操作的右结合性

与下标和解引用操作符一样，赋值操作也返回左值。同理，只要被赋值的每个操作数都具有相同的通用类型，C++语言允许将这多个赋值操作写在一个表达式中：

```
int ival, jval;
ival = jval = 0; // ok: each assigned 0
```

与其他二元操作符不同，赋值操作具有右结合特性。当表达式含有多个赋值操作符时，从右向左结合。上述表达式，将右边赋值操作的结果（也就是 `jval`）赋给 `ival`。多个赋值操作中，各对象必须具有相同的数据类型，或者具有可转换（5.12 节）为同一类型的数据类型：

```
int ival; int *pval;
ival = pval = 0; // error: cannot assign the value of a pointer to an int
string s1, s2;
s1 = s2 = "OK"; // ok: "OK" converted to string
```

第一个赋值语句是不合法的，因为 `ival` 和 `pval` 是不同类型的对象。虽然 `0` 值恰好都可以赋给这两个对象，但该语句仍然错误。因为问题在于给 `pval` 赋值的结果是一个 `int*` 类型的值，不能将此值赋给 `int` 类型的对象。另一方面，第二个赋值语句则是正确的。字符串字面值可以转换为 `string` 类型，`string` 类型的值可赋给 `s2` 变量。右边赋值操作的结果为 `s2`，再将此结果值赋给 `s1`。

5.4.2 赋值操作具有低优先级

另一种通常的用法，是将赋值操作写在条件表达式中，把赋值操作用作长表达式的一部分。这种做法可缩短程序代码并阐明程序员的意图。例如，下面的循环调用函数 `get_value`，假设该函数返回 `int` 数值，通过循环检查这些返回值，直到获得需要的值为止——这里是 `42`：

```
int i = get_value(); // get_value returns an int
while (i != 42) {
    // do something ...
    i = get_value();
}
```

首先，程序将所获得的第一个值存储在 `i` 中，然后建立循环检查 `i` 的值是否为 `42`，如果不是，则做某些处理。循环中的最后一条语句调用 `get_value()` 返回一个值，然后继续循环。该循环可更简洁地写为：

```
int i;
while ((i = get_value()) != 42) {
    // do something ...
}
```

160

现在，循环条件更清晰地表达了程序员的意图：持续循环直到 `get_value` 返回 42 为止。在循环条件中，将 `get_value` 返回的值赋给 `i`，然后判断赋值的结果是否为 42。



在赋值操作上加圆括号是必需的，因为赋值操作符的优先级低于不等操作符。

如果没有圆括号，操作符 `!=` 的操作数则是调用 `get_value` 返回的值和 42，然后将该操作的结果 `true` 或 `false` 赋给 `i` —— 显然这并不是我们想要的。

谨防混淆相等操作符和赋值操作符

可在条件表达式中使用赋值操作，这个事实往往会带来意外的效果：

```
if (i = 42)
```

此代码是合法的：将 42 赋给 `i`，然后检验赋值的结果。此时，42 为非零值，因此解释为 `true`。其实，程序员的目的显然是想判断 `i` 的值是否为 42：

```
if (i == 42)
```

这种类型的程序错误很难发现。有些（并非全部）编译器会为类似于上述例子的代码提出警告。

习题

习题 5.11 请问每次赋值操作完成后，`i` 和 `d` 的值分别是多少？

```
int i; double d;
d = i = 3.5 ;
i = d = 3.5 ;
```

习题 5.12 解释每个 `if` 条件判断产生什么结果？

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

5.4.3 复合赋值操作符

我们常常在对某个对象做某种操作后，再将操作结果重新赋给该对象。例如，考虑 1.4.2 节的求和程序：

```
int sum = 0;
// sum values from 1 up to 10 inclusive
for (int val = 1; val <= 10; ++val)
    sum += val; // equivalent to sum = sum + val
```

161

C++语言不仅对加法，而且还对其他算术操作符和位操作符提供了这种用法，称为复合赋值操作。复合赋值操作符的一般语法格式为：

```
a op= b;
```

其中，`op=` 可以是下列十个操作符之一：

```
+=   -=   *=   /=   %=   // arithmetic operators
<<= >>=   &=   ^=   |=   // bitwise operators
```

每个复合赋值操作符本质上等价于：

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

这两种语法形式存在一个显著的差别：使用复合赋值操作时，左操作数只计算了一次；而使用相似的长表达式时，该操作数则计算了两次，第一次作为右操作数，而第二次则用做左操作数。除非考虑可能的性能价值，在很多（可能是大部分的）上下文环境里这个差别不是本质性的。

习题

习题 5.13 下列赋值操作是不合法的，为什么？怎样改正？

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

习题 5.14 虽然下列表达式都是合法的，但并不是程序员期望的操作，为什么？怎样修改这些表达式以使其能反映程序员的意图？

```
(a) if ( ptr = retrieve_pointer() != 0 )
(b) if ( ival = 1024 )
(c) ival += ival + 1;
```

5.5 自增和自减操作符

自增（++）和自减（--）操作符为对象加 1 或减 1 操作提供了方便简短的实现方式。它们有前置和后置两种使用形式。到目前为止，我们已经使用过前自增操作，该操作使其操作数加 1，操作结果是修改后（changed）的值。同理，前自减操作使其操作数减 1。这两种操作符的后置形式同样对其操作数加 1（或减 1），但操作后产生操作数原来的、未修改的值作为表达式的结果：

```
int i = 0, j;
j = ++i; // j = 1, i = 1: prefix yields incremented value
j = i++; // j = 1, i = 2: postfix yields unincremented value
```

162 因为前置操作返回加 1 后的值，所以返回对象本身，这是左值。而后置操作返回的则是右值。

建议：只有在必要时才使用后置操作符

有使用 C 语言背景的读者可能会觉得奇怪，为什么要在程序中使用前自增操作。道理很简单：因为前置操作需要做的工作更少，只需加 1 后返回加 1 后的结果即可。而后置操作符则必须先保存操作数原来的值，以便返回未加 1 之前的值作为操作的结果。对于 int 型对象和指针，编译器可优化掉这项额外工作。但是对于更多的复杂迭代器类型，这种额外工作可能会花费更大的代价。因此，养成使用前置操作这个好习惯，就不必操心性能差异的问题。

1. 后置操作符返回未加 1 的值

当我们在单个复合表达式中使用变量的当前值，然后再加 1 时，通常会使用后置的++

和--操作:

```
vector<int> ivec;           // empty vector
int cnt = 10;
// add elements 10...1 to ivec
while (cnt > 0)
    ivec.push_back(cnt--); // int postfix decrement
```

这段程序使用了后置的--操作实现 cnt 减 1。我们希望把 cnt 的值赋给 vector 对象的下一个元素,然后在下次迭代前 cnt 的值减 1。如果在循环中使用前置操作,则是用 cnt 减 1 后的值创建 ivec 的新元素,结果是将 9 至 0 十个元素依次添加到 ivec 中。

2. 在单个表达式中组合使用解引用和自增操作

下面的程序使用了一种非常通用的 C++ 编程模式输出 ivec 的内容:

```
vector<int>::iterator iter = ivec.begin();
// prints 10 9 8 ... 1
while (iter != ivec.end())
    cout << *iter++ << endl; // iterator postfix increment
```



如果程序员对 C++ 和 C 语言都不太熟悉,则常常会弄不清楚表达式 *iter++ 的含义。

由于后自增操作的优先级高于解引用操作,因此 *iter++ 等效于 *(iter++)。子表达式 iter++ 使 iter 加 1,然后返回 iter 原值的副本作为该表达式的结果。因此,解引用操作 * 的操作数是 iter 未加 1 前的副本。

这种用法的根据在于后自增操作返回其操作数原值(没有加 1)的副本。如果返回的是加 1 后的值,则解引用该值将导致错误的结果: ivec 的第一个元素没有输出,并企图对一个多余的元素进行解引用。

建议: 简洁即是美

没有 C 语言基础的 C++ 新手,时常会因精简的表达式而苦恼,特别是像 *iter++ 这类令人困惑的表达式。有经验的 C++ 程序员非常重视简练,他们更喜欢这么写:

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

而不采用下面这种冗长的等效代码:

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

对于初学 C++ 的程序员来说,第二种形式更清晰,因为给迭代器加 1 和获取输出值这两个操作是分开来实现的。但是更多的 C++ 程序员更习惯使用第一种形式。

要不断地研究类似的代码,最后达到一目了然的地步。大部分的 C++ 程序员更喜欢使用简洁的表达式而非冗长的等效表达式。因此, C++ 程序员必须熟悉这种用法。而且,一旦熟悉了这类表达式,我们会发现使用起来更不容易出错。

习题

习题 5.15 解释前自增操作和后自增操作的差别。

习题 5.16 你认为为什么 C++ 不叫做 ++C?

习题 5.17 如果输出 vector 内容的 while 循环使用前自增操作符，那会怎么样?

5.6 箭头操作符

C++ 语言为包含点操作符和解引用操作符的表达式提供了一个同义词：箭头操作符 (->)。点操作符 (1.5.2 节) 用于获取类类型对象的成员：

```
item1.same_isbn(item2); // run the same_isbn member of item1
```

如果有一个指向 Sales_item 对象的指针 (或迭代器)，则在使用点操作符前，需对该指针 (或迭代器) 进行解引用：

```
Sales_item *sp = &item1;
(*sp).same_isbn(item2); // run same_isbn on object to which sp points
```

这里，对 sp 进行解引用以获得指定的 Sales_item 对象。然后使用点操作符调用指定对象的 same_isbn 成员函数。在上述用法中，注意必须用圆括号把解引用括起来，因为解引用的优先级低于点操作符。如果漏掉圆括号，则这段代码的含义就完全不同了：

```
// run the same_isbn member of sp then dereference the result!
*sp.same_isbn(item2); // error: sp has no member named same_isbn
```

164

这个表达式企图获得 sp 对象的 same_isbn 成员。等价于：

```
*(sp.same_isbn(item2)); // equivalent to *sp.same_isbn(item2);
```

然而，sp 是一个没有成员的指针；这段代码无法通过编译。

因为编程时很容易忘记圆括号，而且这类代码又经常使用，所以 C++ 为在点操作符后使用的解引用操作定义了一个同义词：箭头操作符 (->)。假设有一个指向类类型对象的指针 (或迭代器)，下面的表达式相互等价：

```
(*p).foo; // dereference p to get an object and fetch its member named foo
p->foo;    // equivalent way to fetch the foo from the object to which p points
```

具体地，可将 same_isbn 的调用重写为：

```
sp->same_isbn(item2); // equivalent to (*sp).same_isbn(item2)
```

习题

习题 5.18 编写程序定义一个 vector 对象，其每个元素都是指向 string 类型的指针，读取该 vector 对象，输出每个 string 的内容及其相应的长度。

习题 5.19 假设 iter 为 vector<string>::iterator 类型的变量，指出下面哪些表达式是合法的，并解释这些合法表达式的行为。

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

5.7 条件操作符

条件操作符 (conditional operator) 是 C++ 中唯一的三元操作符，它允许将简单的 if-else 判断语句嵌入表达式中。条件操作符的语法格式为：

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

其中，*cond* 是一个条件判断表达式 (1.4.1 节)。条件操作符首先计算 *cond* 的值，如果 *cond* 的值为 0，则条件为 false；如果 *cond* 非 0，则条件为 true。无论如何，*cond* 总是要被计算的。然后，条件为 true 时计算 *expr1*，否则计算 *expr2*。和逻辑与、逻辑或 (&& 和 ||) 操作符一样，条件操作符保证了上述操作数的求解次序。*expr1* 和 *expr2* 中只有一个表达式被计算。下面的程序说明了条件操作符的用法：

```
int i = 10, j = 20, k = 30;
// if i > j then maxVal = i else maxVal = j
int maxVal = i > j ? i : j;
```

165

1. 避免条件操作符的深度嵌套

可以使用一组嵌套的条件操作符求出三个变量的最大值，并将最大值赋给 max：

```
int max = i > j
    ? i > k ? i : k
    : j > k ? j : k;
```

我们也可以下面更长却更简单的比较语句实现相同的功能：

```
int max = i;
if (j > max)
    max = j;
if (k > max)
    max = k;
```

2. 在输出表达式中使用条件操作符

条件操作符的优先级相当低。当我们要在一个更大的表达式中嵌入条件表达式时，通常必须用圆括号把条件表达式括起来。例如，经常使用条件操作符根据一定的条件输出一个或另一个值，在输出表达式中，如果不严格使用圆括号将条件操作符括起来，将会得到意外的结果：

```
cout << (i < j ? i : j); // ok: prints larger of i and j
cout << (i < j) ? i : j; // prints 1 or 0!
cout << i < j ? i : j;   // error: compares cout to int
```

第二个表达式比较有趣：它将 *i* 和 *j* 的比较结果视为 << 操作符的操作数，输出 1 或 0。<< 操作符返回 cout 值，然后将返回结果作为条件操作符的判断条件。也就是，第二个表达式等效于：

```
cout << (i < j); // prints 1 or 0
cout ? i : j;     // test cout and then evaluate i or j
```

```
// depending on whether cout evaluates to true or false
```

习题

习题 5.20 编写程序提示用户输入两个数，然后报告哪个数比较小。

习题 5.21 编写程序处理 `vector<int>` 对象的元素：将每个奇数值元素用该值的两倍替换。

166

5.8 sizeof 操作符

`sizeof` 操作符的作用是返回一个对象或类型名的长度，返回值的类型为 `size_t` (3.5.2 节)，长度的单位是字节 (2.1 节)。`sizeof` 表达式的结果是编译时常量，该操作符有以下三种语法形式：

```
sizeof (type name);
sizeof (expr);
sizeof expr;
```

将 `sizeof` 应用在表达式 `expr` 上，将获得该表达式的结果的类型长度：

```
Sales_item item, *p;
// three ways to obtain size required to hold an object of type Sales_item
sizeof(Sales_item); // size required to hold an object of type Sales_item
sizeof item; // size of item's type, e.g., sizeof(Sales_item)
sizeof *p; // size of type to which p points, e.g., sizeof(Sales_item)
```

将 `sizeof` 用于 `expr` 时，并没有计算表达式 `expr` 的值。特别是在 `sizeof *p` 中，指针 `p` 可以持有一个无效地址，因为不需要对 `p` 做解引用操作。

使用 `sizeof` 的结果部分地依赖所涉及的类型：

- 对 `char` 类型或值为 `char` 类型的表达式做 `sizeof` 操作保证得 1。
- 对引用类型做 `sizeof` 操作将返回存放此引用类型对象所需的内存空间大小。
- 对指针做 `sizeof` 操作将返回存放指针所需的内存大小；注意，如果要获取该指针所指向对象的大小，则必须对该指针进行解引用。
- 对数组做 `sizeof` 操作等效于将对其元素类型做 `sizeof` 操作的结果乘上数组元素的个数。

因为 `sizeof` 返回整个数组在内存中的存储长度，所以用 `sizeof` 数组的结果除以 `sizeof` 其元素类型的结果，即可求出数组元素的个数：

```
// sizeof(ia)/sizeof(*ia) returns the number of elements in ia
int sz = sizeof(ia)/sizeof(*ia);
```

习题

习题 5.22 编写程序输出每种内置类型的长度。

习题 5.23 预测下列程序的输出，并解释你的理由。然后运行该程序，输出的结果和你预测的一样吗？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;
```

167

5.9 逗号操作符

逗号表达式是一组由逗号分隔的表达式，这些表达式从左向右计算。逗号表达式的结果是其最右边表达式的值。如果最右边的操作数是左值，则逗号表达式的值也是左值。此类表达式通常用于 for 循环：

```
int cnt = ivec.size();
// add elements from size... 1 to ivec
for(vector<int>::size_type ix = 0;
    ix != ivec.size(); ++ix, --cnt)
    ivec[ix] = cnt;
```

上述的 for 语句在循环表达式中使 ix 自增 1 而 cnt 自减 1。每次循环均要修改 ix 和 cnt 的值。当检验 ix 的条件判断成立时，程序将下一个元素重新设置为 cnt 的当前值。

习题

习题 5.24 本节的程序与 5.5 节在 vector 对象中添加元素的程序类似。两段程序都使用递减的计数器生成元素的值。本程序中，我们使用了前自减操作，而 5.5 节的程序则使用了后自减操作。解释为什么一段程序中使用前自减操作而在另一段程序中使用后自减操作。

5.10 复合表达式的求值

含有两个或更多操作符的表达式称为**复合表达式**（compound expression）。在复合表达式中，操作数和操作符的结合方式决定了整个表达式的值。表达式的结果会因为操作符和操作数的分组结合方式的不同而不同。

操作数的分组结合方式取决于操作符的优先级和结合性。也就是说，优先级和结合性决定了表达式的哪个部分用作哪个操作符的操作数。如果程序员不想考虑这些规则，可以在复合表达式中使用圆括号强制实现某个特殊的分组。



优先级规定的是操作数的结合方式，但并没有说明操作数的计算顺序。在大多数情况下，操作数一般以最方便的次序求解。

5.10.1 优先级

表达式的值取决于其子表达式如何分组。例如，下面的表达式，如果纯粹从左向右计算，结果为 20：

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

想像中其他可能的结果包括 9、14 和 36。在 C++ 中，该表达式的值应为 14。

乘法和除法的优先级高于加法操作，于是它们的操作数先于加法操作的操作数计算。但乘法

和除法的优先级相同。当操作符的优先级相同时，由其结合性决定求解次序。算术操作具有左结合性，这意味着它们从左向右结合。因此上面表达式等效于：

```
int temp = 3 * 4;           // 12
int temp2 = temp / 2;       // 6
int temp3 = temp2 + 6;      // 12
int result = temp3 + 2;     // 14
```

圆括号凌驾于优先级之上

我们可使用圆括号推翻优先级的限制。使用圆括号的表达式将用圆括号括起来的子表达式视为独立单元先计算，其他部分则以普通的优先级规则处理。例如，下面的程序在前述表达式上添加圆括号，强行更改其操作次序，可能得到四种结果：

```
// parentheses on this expression match default precedence/associativity
cout << ((6 + ((3 * 4) / 2)) + 2) << endl; // prints 14
// parentheses result in alternative groupings
cout << (6 + 3) * (4 / 2 + 2) << endl;      // prints 36
cout << ((6 + 3) * 4) / 2 + 2 << endl;      // prints 20
cout << 6 + 3 * 4 / (2 + 2) << endl;        // prints 9
```

我们已经通过前面的例子了解了优先级规则如何影响程序的正确性。例如，考虑 5.5 节第二个建议框中描述的表达式：

```
*iter++;
```

其中，++的优先级高于*操作符，这就意味着 iter++先结合。而操作符*的操作数是 iter 做了自增操作后的结果。如果我们希望对 iter 所指向的值做自增操作，则必须使用圆括号强制实现我们的目的：

```
(*iter)++; // increment value to which iter refers and yield unincremented value
```

圆括号指明操作符*的操作数是 iter，然后表达式以*iter 作为++操作符的操作数。

另一个例子，回顾一下 5.4.2 节中的 while 循环条件：

```
while ((i = get_value()) != 42) {
```

赋值操作上的圆括号是必需的，这样才能实现预期的操作：将 get_value 的返回值赋给 i，然后检查刚才赋值的结果是否为 42。如果赋值操作上没有加圆括号，结果将是先判断 get_value 的返回值是否为 42，然后将判断结果 true 或 false 值赋给 i，这意味着 i 的值只能是 1 或 0。

169

5.10.2 结合性

结合性规定了具有相同优先级的操作符如何分组。我们已经遇到过涉及结合性的例子。其中之一使用了赋值操作的右结合性，这个特性允许将多个赋值操作串接起来：

```
ival = jval = kval = lval // right associative
(ival = (jval = (kval = lval))) // equivalent, parenthesized version
```

该表达式首先将 lval 赋给 kval，然后将 kval 的值赋给 jval，最后将 jval 的值再赋给 ival。

另一方面，算术操作符为左结合。表达式

```
ival * jval / kval * lval          // left associative
(((ival * jval) / kval) * lval) // equivalent, parenthesized version
```

先对 ival 和 jval 做乘法操作，然后乘积除以 kval，最后再将其商与 lval 相乘。

表 5-4 按照优先级顺序列出了 C++ 的全部操作符。该表以双横线分割成不同的段，每段内各个操作符的优先级相同，且都高于后面各段中的操作符。例如，前自增操作符和解引用操作符的优先级相同，它们的优先级都比算术操作符或关系操作符高。此表中大部分操作符已经介绍过，而少数未介绍的操作符将在后续章节中学习。

表 5-4 操作符的优先级				
操作符及其结合性		功 能	用 法	参见页面
L	::	全局作用域	::name	p.389
L	::	类作用域	class::name	p.73
L	::	名字空间作用域	namespace::name	p.68
L	.	成员选择	object.member	p.20
L	->	成员选择	pointer->member	p.142
L	[]	下标	variable[expr]	p.99
L	()	函数调用	name(expr_list)	p.20
L	()	类型构造	type(expr_list)	p.393
R	++	后自增操作	lvalue++	p.140
R	--	后自减操作	lvalue--	p.140
R	typeid	类型 ID	typeid(type)	p.649
R	typeid	运行时类型 ID	typeid(expr)	p.649
R	显式强制类型转换	类型转换	cast_name<type>(expr)	p.158
R	sizeof	对象的大小	sizeof expr	p.144
R	sizeof	类型的大小	sizeof(type)	p.144
R	++	前自增操作	++lvalue	p.140
R	--	前自减操作	--lvalue	p.140
R	~	位求反	~expr	p.134
R	!	逻辑非	!expr	p.131
R	-	一元负号	-expr	p.129
R	+	一元正号	+expr	p.129
R	*	解引用	*expr	p.104
R	&	取地址	&expr	p.100
R	()	类型转换	(type)expr	p.160
R	new	创建对象	new type	p.150
R	delete	释放对象	delete expr	p.152
R	delete[]	释放数组	delete []expr	p.119
L	->*	指向成员操作的指针	ptr->*ptr_to_member	p.655
L	.*	指向成员操作的指针	obj.*ptr_to_member	p.655
L	*	乘法	expr * expr	p.129
L	/	除法	expr / expr	p.129
L	%	求模 (求余)	expr % expr	p.129
L	+	加法	expr + expr	p.129
L	-	减法	expr - expr	p.129

(续)

操作符及其结合性		功 能	用 法	参见页面
L	<<	位左移	expr << expr	p.134
L	>>	位右移	expr >> expr	p.134
L	<	小于	expr < expr	p.131
L	<=	小于或等于	expr <= expr	p.131
L	>	大于	expr > expr	p.131
L	>=	大于或等于	expr >= expr	p.131
L	==	相等	expr == expr	p.131
L	!=	不等	expr != expr	p.131
L	&	位与	expr & expr	p.134
L	^	位异或	expr ^ expr	p.134
L		位或	expr expr	p.134
L	&&	逻辑与	expr && expr	p.131
L		逻辑或	expr expr	p.131
R	?:	条件操作	expr ? expr : expr	p.143
R	=	赋值操作	lvalue = expr	p.137
R	*, /=, %=,	复合赋值操作	lvalue += expr 等等	p.137
R	+=, -=,			p.137
R	<<=, >>=,			p.137
R	&=, =, ^=			p.137
R	throw	抛出异常	throw expr	p.186
L	,	逗号	expr, expr	p.145

171

习题

习题 5.25 根据表 5-4 的内容, 在下列表达式中添加圆括号说明其操作数分组的顺序(即计算顺序):

- (a) ! ptr == ptr->next
- (b) ch = buf[bp++] != '\n'

习题 5.26 习题 5.25 中的表达式的计算次序与你的意图不同, 给它们加上圆括号使其以你所希望的操作次序求解。

习题 5.27 由于操作符优先级的问題, 下列表达式编译失败。请参照表 5-4 解释原因, 应该如何改正?

```
string s = "word";  
// add an 's' to the end, if the word doesn't already end in 's'  
string pl = s + s[s.size() - 1] == 's' ? "" : "s" ;
```

5.10.3 求值顺序

在 5.2 节中, 我们讨论了&&和||操作符计算其操作数的次序: 当且仅当其右操作数确实影响了整个表达式的值时, 才计算这两个操作符的右操作数。根据这个原则, 可编写如下代码:

```
// iter only dereferenced if it isn't at end  
while (iter != vec.end() && *iter != some_val)
```

C++中，规定了操作数计算顺序的操作符还有条件(?:)和逗号操作符。除此之外，其他操作符并未指定其操作数的求值顺序。

例如，表达式

```
f1() * f2();
```

在做乘法操作之前，必须调用 f1 函数和 f2 函数，毕竟其调用结果要相乘。然而，我们却无法知道到底是先调用 f1 还是先调用 f2。



其实，以什么次序求解操作数通常没有多大关系。只有当操作符的两个操作数涉及到同一个对象，并改变其值时，操作数的计算次序才会影响结果。

如果一个子表达式修改了另一个子表达式的操作数，则操作数的求解次序就变得相当重要：

```
// oops! language does not define order of evaluation
```

```
if (ia[index++] < ia[index])
```

172

此表达式的行为没有明确定义。问题在于：<操作符的左右操作数都使用了 index 变量，但是，左操作数更改了该变量的值。假设 index 初值为 0，编译器可以用下面两种方式之一求该表达式的值：

```
if (ia[0] < ia[0]) // execution if rhs is evaluated first
```

```
if (ia[0] < ia[1]) // execution if lhs is evaluated first
```

可以假设程序员希望先求左操作数的值，因此 index 的值加 1。如果是这样的话，比较 ia[0] 和 ia[1] 的值。然而，C++ 语言不能确保从左到右的计算次序。事实上，这类表达式的行为没有明确定义。一种实现可能是先计算右操作数，于是 ia[0] 与自己做比较，要不然就是做完全不同的操作。

建议：复合表达式的处理

初学 C 和 C++ 的程序员一般很难理解求值顺序、优先级和结合性规则。误解表达式和操作数如何求解将导致大量的程序错误。此外，除非程序员已经完全理解了相关规则，否则这类错误很难发现，因为仅靠阅读程序是无法排除这些错误的。

下面两个指导原则有助于处理复合表达式：

- (1) 如果有怀疑，则在表达式上按程序逻辑要求使用圆括号强制操作数的组合。
- (2) 如果要修改操作数的值，则不要在同一个语句的其他地方使用该操作数。如果必须使用改变的值，则把该表达式分割成两个独立语句：在一个语句中改变该操作数的值，再在下一个语句使用它。

第二个规则有一个重要的例外：如果一个子表达式修改操作数的值，然后将该子表达式的结果用于另一个子表达式，这样则是安全的。例如，`*++iter` 表达式的自增操作修改了 iter 的值，然后将 iter (修改后) 的值用作 * 操作符的操作数。对于这个表达式或其他类似的表达式，其操作数的计算次序无关紧要。而为了计算更复杂的表达式，改变操作数值的子表达式必须首先计算。这种方法很常用，不会产生什么问题。



一个表达式里，不要在两个或更多的子表达式中对同一对象做自增或自减操作。

以一种安全而且独立于机器的方式重写上述比较两个数组元素的程序：

```
if (ia[index] < ia[index + 1]) {
    // do whatever
}
++index;
```

173 现在，两个操作数的值不会相互影响。

习题

习题 5.28 除了逻辑与和逻辑或外，C++没有明确定义二元操作符的求解次序，编译器可自由地提供最佳的实现方式。只能在“实现效率”和程序语言使用中“潜在的缺陷”之间寻求平衡。你认为这可以接受吗？说出你的理由。

习题 5.29 假设 ptr 指向类类型对象，该类拥有一个名为 ival 的 int 型数据成员，vec 是保存 int 型元素的 vector 对象，而 ival、jval 和 kval 都是 int 型变量。请解释下列表达式的行为，并指出哪些（如果有的话）可能是不正确的，为什么？如何改正？

```
(a) ptr->ival != 0          (b) ival != jval < kval
(c) ptr != 0 && *ptr++      (d) ival++ && ival
(e) vec[ival++] <= vec[ival]
```

5.11 new 和 delete 表达式

4.3.1 节介绍了如何使用 new 和 delete 表达式动态创建和释放数组，这两种表达式也可用于动态创建和释放单个对象。

定义变量时，必须指定其数据类型和名字。而动态创建对象时，只需指定其数据类型，而不必为该对象命名。取而代之的是，new 表达式返回指向新创建对象的指针，我们通过该指针来访问此对象：

```
int i;                // named, uninitialized int variable
int *pi = new int;    // pi points to dynamically allocated,
                      // unnamed, uninitialized int
```

这个 new 表达式在自由存储区中分配创建了一个整型对象，并返回此对象的地址，并用该地址初始化指针 pi。

1. 动态创建对象的初始化

动态创建的对象可用初始化变量的方式实现初始化：

```
int i(1024);           // value of i is 1024
int *pi = new int(1024); // object to which pi points is 1024
string s(10, '9');     // value of s is "9999999999"
string *ps = new string(10, '9'); // *ps is "9999999999"
```

C++使用直接初始化 (direct-initialization) 语法规则 (2.3.3 节) 初始化动态创建的对象。如果提供了初值, new 表达式分配到所需要的内存后, 用给定的初值初始化该内存空间。在本例中, pi 所指向的新创建对象将被初始化为 1024, 而 ps 所指向的对象则初始化为十个 9 的字符串。

174

2. 动态创建对象的默认初始化

如果不提供显式初始化, 动态创建的对象与在函数内定义的变量初始化方式相同 (2.3.4 节)。对于类类型的对象, 用该类的默认构造函数初始化; 而内置类型的对象则无初始化。

```
string *ps = new string;    // initialized to empty string
int *pi = new int;         // pi points to an uninitialized int
```

通常, 除了对其赋值之外, 对未初始化的对象所关联的值的任何使用都是没有定义的。



最佳
实践

正如我们 (几乎) 总是要初始化定义为变量的对象一样, 在动态创建对象时, (几乎) 总是对它做初始化也是一个好办法。

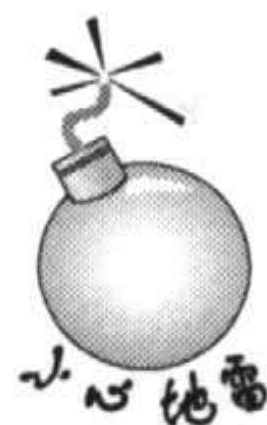
同样也可对动态创建的对象做值初始化 (value-initialize) (3.3.1 节):

```
string *ps = new string(); // initialized to empty string
int *pi = new int();       // pi points to an int value-initialized to 0
cls *pc = new cls();       // pc points to a value-initialized object of type cls
```

以上表明程序员想通过在类型名后面使用一对内容为空的圆括号对动态创建的对象做值初始化。内容为空的圆括号表示虽然要做初始化, 但实际上并未提供特定的初值。对于提供了默认构造函数的类类型 (例如 string), 没有必要对其对象进行值初始化: 无论程序是明确地不初始化还是要求进行值初始化, 都会自动调用其默认构造函数初始化该对象。而对于内置类型或没有定义默认构造函数的类型, 采用不同初始化方式则有显著的差别:

```
int *pi = new int;           // pi points to an uninitialized int
int *pi = new int();         // pi points to an int value-initialized to 0
```

第一个语句的 int 型变量没有初始化, 而第二个语句的 int 型变量则被初始化为 0。



小心地雷

值初始化的 () 语法必须置于类型名后面, 而不是变量后。正如我们将要学习的 7.4 节的例子:

```
int x(); // does not value initialize x
```

这个语句声明了一个名为 x、没有参数而且返回 int 值的函数。

3. 耗尽内存

尽管现代机器的内存容量越来越大, 但是自由存储区总有可能被耗尽。如果程序用完了所有可用的内存, new 表达式就有可能失败。如果 new 表达式无法获取需要的内存空间, 系统将抛出名为 bad_alloc 的异常。我们将在 6.13 节介绍如何抛出异常。

175

4. 撤销动态创建的对象

动态创建的对象用完后, 程序员必须显式地将该对象占用的内存返回给自由存储区。C++ 提供了 delete 表达式释放指针所指向的地址空间。

```
delete pi;
```

该命令释放 `pi` 指向的 `int` 型对象所占用的内存空间。



如果指针指向不是用 `new` 分配的内存地址, 则在该指针上使用 `delete` 是不合法的。

C++ 没有明确定义如何释放指向不是用 `new` 分配的内存地址的指针。下面提供了一些安全的和不安全的 `delete` 表达式。

```
int i;
int *pi = &i;
string str = "dwarves";
double *pd = new double(33);
delete str; // error: str is not a dynamic object
delete pi; // error: pi refers to a local
delete pd; // ok
```

值得注意的是: 编译器可能会拒绝编译 `str` 的 `delete` 语句。编译器知道 `str` 并不是一个指针, 因此会在编译时就能检查出这个错误。第二个错误则比较隐蔽: 通常来说, 编译器不能断定一个指针指向什么类型的对象, 因此尽管这个语句是错误的, 但在大部分编译器上仍能通过。

5. 零值指针的删除

如果指针的值为 0, 则在其上做 `delete` 操作是合法的, 但这样做没有任何意义:

```
int *ip = 0;
delete ip; // ok: always ok to delete a pointer that is equal to 0
```

C++ 保证: 删除 0 值的指针是安全的。

6. 在 `delete` 之后, 重设指针的值

执行语句

```
delete p;
```

后, `p` 变成没有定义。在很多机器上, 尽管 `p` 没有定义, 但仍然存放了它之前所指向对象的地址, 然而 `p` 所指向的内存已经被释放, 因此 `p` 不再有效。

删除指针后, 该指针变成悬垂指针 (dangling pointer)。悬垂指针指向曾经存放对象的内存, 但该对象已经不再存在了。悬垂指针往往导致程序错误, 而且很难检测出来。

176



一旦删除了指针所指向的对象, 立即将指针置为 0, 这样就非常清楚地表明指针不再指向任何对象。

7. `const` 对象的动态分配和回收

C++ 允许动态创建 `const` 对象:

```
// allocate and initialize a const object
const int *pci = new const int(1024);
```

与其他常量一样, 动态创建的 `const` 对象必须在创建时初始化, 并且一经初始化, 其值就不能再修改。上述 `new` 表达式返回指向 `int` 型 `const` 对象的指针。与其他 `const` 对象的地址一样, 由于 `new` 返回的地址上存放的是 `const` 对象, 因此该地址只能赋给指向 `const` 的指针。

对于类类型的 `const` 动态对象，如果该类提供了默认的构造函数，则此对象可隐式初始化：

```
// allocate default initialized const empty string
const string *pcs = new const string;
```

`new` 表达式没有显式初始化 `pcs` 所指向的对象，而是隐式地将 `pcs` 所指向的对象初始化为空的 `string` 对象。内置类型对象或未提供默认构造函数的类类型对象必须显式初始化。

警告：动态内存的管理容易出错

下面三种常见的程序错误都与动态内存分配相关：

(1) 删除 (`delete`) 指向动态分配内存的指针失败，因而无法将该块内存返还给自由存储区。删除动态分配内存失败称为“内存泄漏 (`memory leak`)”。内存泄漏很难发现，一般需等应用程序运行了一段时间后，耗尽了所有内存空间时，内存泄漏才会显露出来。

(2) 读写已删除的对象。如果删除指针所指向的对象之后，将指针置为 0 值，则比较容易检测出这类错误。

(3) 对同一个内存空间使用两次 `delete` 表达式。当两个指针指向同一个动态创建的对象，删除时就会发生错误。如果在其中一个指针上做 `delete` 运算，将该对象的内存空间返还给自由存储区，然后接着 `delete` 第二个指针，此时则自由存储区可能会被破坏。

操纵动态分配的内存时，很容易发生上述错误，但这些错误却难以跟踪和修正。

177

8. 删除 `const` 对象

尽管程序员不能改变 `const` 对象的值，但可撤销对象本身。如同其他动态对象一样，`const` 动态对象也是使用删除指针来释放的：

```
delete pci; // ok: deletes a const object
```

即使 `delete` 表达式的操作数是指向 `int` 型 `const` 对象的指针，该语句同样有效地回收 `pci` 所指向的内容。

习题

习题 5.30 下列语句哪些（如果有的话）是非法的或错误的？

- (a) `vector<string> svec(10);`
- (b) `vector<string> *pvec1 = new vector<string>(10);`
- (c) `vector<string> **pvec2 = new vector<string>[10];`
- (d) `vector<string> *pv1 = &svec;`
- (e) `vector<string> *pv2 = pvec1;`
- (f) `delete svec;`
- (g) `delete pvec1;`
- (h) `delete [] pvec2;`
- (i) `delete pv1;`
- (j) `delete pv2;`

5.12 类型转换

表达式是否合法取决于操作数的类型，而且合法的表达式其含义也由其操作数类型决定。但是，在 C++ 中，某些类型之间存在相关的依赖关系。若两种类型相关，则可在需要某种类型的操作数位置上，使用该类型的相关类型对象或值。如果两个类型之间可以相互转换（conversion），则称这两个类型相关。

考虑下列例子：

```
int ival = 0;
ival = 3.541 + 3; // typically compiles with a warning
```

ival 的值为 6。

首先做加法操作，其操作数是两个不同类型的值：3.541 是 double 型的字面值常量，而 3 则是 int 型的字面值常量。C++ 并不是把两个不同类型的值直接加在一起，而是提供了一组转换规则，以便在执行算术操作之前，将两个操作数转换为同一种数据类型。这些转换规则由编译器自动执行，无需程序员介入——有时甚至不需要程序员了解。因此，它们也被称为隐式类型转换（implicit type conversion）。

C++ 定义了算术类型之间的内置转换以尽可能防止精度损失。通常，如果表达式的操作数分别为整型和浮点型，则整型的操作数被转换为浮点型。本例中，整数 3 被转换为 double 类型，然后执行浮点类型的加法操作，得 double 类型的结果 6.541。

下一步是将 double 类型的值赋给 int 型变量 ival。在赋值操作中，因为不可能更改左操作数对象的类型，因此左操作数的类型占主导地位。如果赋值操作的左右操作数类型不相同，则右操作数会被转换为左边的类型。本例中，double 型的加法结果转换为 int 型。double 向 int 的转换自动按截尾形式进行，小数部分被舍弃。于是 6.541 变成 6，然后赋给 ival。因为从 double 到 int 的转换会导致精度损失，因此大多数编译器会给出警告。例如，本书所用的测试例程的编译器给出如下警告：

```
warning: assignment to 'int' from 'double'
```

为了理解隐式类型转换，我们需要知道它们在什么时候发生，以及可能出现什么类型的转换。

5.12.1 何时发生隐式类型转换

编译器在必要时将类型转换规则应用到内置类型和类类型的对象上。在下列情况下，将发生隐式类型转换：

- 在混合类型的表达式中，其操作数被转换为相同的类型：

```
int ival;
double dval;
ival >= dval // ival converted to double
```

- 用作条件的表达式被转换为 bool 类型：

```
int ival;
if (ival) // ival converted to bool
while (cin) // cin converted to bool
```

条件操作符(?:)中的第一个操作数以及逻辑非(!)、逻辑与(&&)和逻辑或(||)的操作数都是条件表达式。出现在 if、while、for 和 do while 语句中的同样也是条件表达式(其中 do while 将在第 6 章中学习)。

- 用一表达式初始化某个变量, 或将一表达式赋值给某个变量, 则该表达式被转换为该变量的类型:

```
int ival = 3.14; // 3.14 converted to int
int *ip;
ip = 0; // the int 0 converted to a null pointer of type int *
```

另外, 在函数调用中也可能发生隐式类型转换, 我们将在第 7 章学习这方面的内容。

179

5.12.2 算术转换

C++语言为内置类型提供了一组转换规则, 其中最常用的是**算术转换(arithmetic conversion)**。算术转换保证在执行操作之前, 将二元操作符(如算术或逻辑操作符)的两个操作数转换为同一类型, 并使表达式的值也具有相同的类型。

算术转换规则定义了一个类型转换层次, 该层次规定了操作数应按什么次序转换为表达式中最宽的类型。在包含多种类型的表达式中, 转换规则要确保计算值的精度。例如, 如果一个操作数的类型是 long double, 则无论另一个操作数是什么类型, 都将被转换为 long double。

最简单的转换为**整型提升(integral promotion)**: 对于所有比 int 小的整型, 包括 char、signed char、unsigned char、short 和 unsigned short, 如果该类型的所有可能的值都能包容在 int 内, 它们就会被提升为 int 型, 否则, 它们将被提升为 unsigned int。如果将 bool 值提升为 int, 则 false 转换为 0, 而 true 则转换为 1。

1. 有符号与无符号类型之间的转换

若表达式中使用了无符号(unsigned)数值, 所定义的转换规则需保护操作数的精度。unsigned 操作数的转换依赖于机器中整型的相对大小, 因此, 这类转换本质上依赖于机器。

包含 short 和 int 类型的表达式, short 类型的值转换为 int。如果 int 型足够表示所有 unsigned short 型的值, 则将 unsigned short 转换为 int, 否则, 将两个操作数均转换为 unsigned int。例如, 如果 short 用半字表示而 int 用一个字表示, 则所有 unsigned 值都能包容在 int 内, 在这种机器上, unsigned short 转换为 int。

long 和 unsigned int 的转换也是一样的。只要机器上的 long 型足够表示 unsigned int 型的所有值, 就将 unsigned int 转换为 long 型, 否则, 将两个操作数均转换为 unsigned long。

在 32 位的机器上, long 和 int 型通常用一个字长表示, 因此当表达式包含 unsigned int 和 long 两种类型, 其操作数都应转换为 unsigned long 型。

对于包含 signed 和 unsigned int 型的表达式, 其转换可能出乎我们的意料。表达式中的 signed 型数值会被转换为 unsigned 型。例如, 比较 int 型和 unsigned int 型的简单变量, 系统首先将 int 型数值转换为 unsigned int 型, 如果 int 型的值恰好为负数, 其结果将以 2.1.1 节介绍的方法转换, 并带来该节描述的所有副作用。

2. 理解算术转换

180

研究大量例题是帮助理解算术转换的最好方法。下面大部分例题中，要么是将操作数转换为表达式中的最大类型，要么是在赋值表达式中将右操作数转换为左操作数的类型。

```

bool    flag;        char    cval;
short   sval;        unsigned short usval;
int      ival;        unsigned int  uival;
long     lval;        unsigned long ulval;
float    fval;        double    dval;
3.14159L + 'a'; // promote 'a' to int, then convert to long double
dval + ival;    // ival converted to double
dval + fval;    // fval converted to double
ival = dval;    // dval converted (by truncation) to int
flag = dval;    // if dval is 0, then flag is false, otherwise true
cval + fval;    // cval promoted to int, that int converted to float
sval + cval;    // sval and cval promoted to int
cval + lval;    // cval converted to long
ival + ulval;   // ival converted to unsigned long
usval + ival;   // promotion depends on size of unsigned short and int
uival + lval;   // conversion depends on size of unsigned int and long

```

第一个加法操作的小写字母 'a' 是一个 char 类型的字符常量，正如我们在 2.1.1 节介绍的，它是一个数值。字母 'a' 表示的数值取决于机器字符集。在 ASCII 机器中，字母 'a' 的值为 97。将 'a' 与 long double 型数据相加时，char 型的值被提升为 int 型，然后将 int 型转换为 long double 型，转换后的值再与 long double 型字面值相加。另一个有趣的现象是最后两个表达式都包含 unsigned 数值。

5.12.3 其他隐式转换

1. 指针转换

在使用数组时，大多数情况下数组都会自动转换为指向第一个元素的指针：

```

int ia[10];    // array of 10 ints
int* ip = ia;  // convert ia to pointer to first element

```

不将数组转换为指针的例外情况有：数组用作取地址 (&) 操作符的操作数或 sizeof 操作符的操作数时，或用数组对数组的引用进行初始化时，不会将数组转换为指针。我们将在 7.2.4 节学习如何定义指向数组的引用（或指针）。

181

C++ 还提供了另外两种指针转换：指向任意数据类型的指针都可转换为 void* 类型；整型数值常量 0 可转换为任意指针类型。

2. 转换为 bool 类型

算术值和指针值都可以转换为 bool 类型。如果指针或算术值为 0，则其 bool 值为 false，而其他值则为 true：

```

if (cp) /* ... */    // true if cp is not zero
while (*cp) /* ... */ // dereference cp and convert resulting char to bool

```

这里，if 语句将 cp 的非零值转换为 true。while 语句则对 cp 进行解引用，操作结果产生一个

char 型的值。空字符 (null) 具有 0 值，被转换为 false，而其他字符值则转换为 true。

3. 算术类型与 bool 类型的转换

可将算术对象转换为 bool 类型，bool 对象也可转换为 int 型。将算术类型转换为 bool 型时，零转换为 false，而其他值则转换为 true。将 bool 对象转换为算术类型时，true 变成 1，而 false 则为 0：

```
bool b = true;
int ival = b;    // ival == 1
double pi = 3.14;
bool b2 = pi;    // b2 is true
pi = false;      // pi == 0
```

4. 转换与枚举类型

C++ 自动将枚举类型 (2.7 节) 的对象或枚举成员 (enumerator) 转换为整型，其转换结果可用于任何要求使用整数值的**地方。例如，用于算术表达式：

```
// point2d is 2, point2w is 3, point3d is 3, point3w is 4
enum Points { point2d = 2, point2w,
              point3d = 3, point3w };
const size_t array_size = 1024;
// ok: pt2w promoted to int
int chunk_size = array_size * pt2w;
int array_3d = array_size * point3d;
```

将 enum 对象或枚举成员提升为什么类型由机器定义，并且依赖于枚举成员的最大值。无论其最大值是什么，enum 对象或枚举成员至少提升为 int 型。如果 int 型无法表示枚举成员的最大值，则提升到能表示所有枚举成员值的、大于 int 型的最小类型 (unsigned int、long 或 unsigned long)。

5. 转换为 const 对象

当使用非 const 对象初始化 const 对象的引用时，系统将非 const 对象转换为 const 对象。此外，还可以将非 const 对象的地址 (或非 const 指针) 转换为指向相关 const 类型的指针：

```
int i;
const int ci = 0;
const int &j = i;    // ok: convert non-const to reference to const int
const int *p = &ci; // ok: convert address of non-const to address of a const
```

6. 由标准库类型定义的转换

类类型可以定义由编译器自动执行的类型转换。迄今为止，我们使用过的标准库类型中，有一个重要的类型转换。从 istream 中读取数据，并将此表达式作为 while 循环条件：

```
string s;
while (cin >> s)
```

这里隐式使用了 IO 标准库定义的类型转换。在与此类似的条件中，求解表达式 cin >> s，即读 cin。无论读入是否成功，该表达式的结果都是 cin。

while 循环条件应为 bool 类型的值，但此时给出的却是 istream 类类型的值，于是 istream 类型的值应转换为 bool 类型。将 istream 类型转换为 bool 类型意味着要检验流的状态。如果

最后一次读 `cin` 的尝试是成功的，则流的状态将导致上述类型转换为 `bool` 类型后获得 `true` 值——`while` 循环条件成立。如果最后一次尝试失败，比如说已经读到文件尾了，此时将 `istream` 类型转换为 `bool` 类型后得 `false`，`while` 循环条件不成立。

习题

习题 5.31 根据 5.12.2 节的变量定义，解释在计算下列表达式的过程中发生了什么类型转换？

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

记住，你可能需要考虑操作符的结合性，以便在表达式含有多个操作符的情况下确定答案。

5.12.4 显式转换

显式转换也称为强制类型转换（`cast`），包括以下列名字命名的强制类型转换操作符：

`static_cast`、`dynamic_cast`、`const_cast` 和 `reinterpret_cast`。



183

虽然有时候确实需要强制类型转换，但是它们本质上是非常危险的。

5.12.5 何时需要强制类型转换

因为要覆盖通常的标准转换，所以需显式使用强制类型转换。下面的复合赋值：

```
double dval;
int ival;
ival *= dval; // ival = ival * dval
```

为了与 `dval` 做乘法操作，需将 `ival` 转换为 `double` 型，然后将乘法操作的 `double` 型结果截尾为 `int` 型，再赋值给 `ival`。为了去掉将 `ival` 转换为 `double` 型这个不必要的转换，可通过如下强制将 `dval` 转换为 `int` 型：

```
ival *= static_cast<int>(dval); // converts dval to int
```

显式使用强制类型转换的另一个原因是：可能存在多种转换时，需要选择一种特定的类型转换。我们将在第 14 章中详细讨论这种情况。

5.12.6 命名的强制类型转换

命名的强制类型转换符号的一般形式如下：

```
cast-name<type>(expression);
```

其中 `cast-name` 为 `static_cast`、`dynamic_cast`、`const_cast` 和 `reinterpret_cast` 之一，`type` 为转换的目标类型，而 `expression` 则是被强制转换的值。强制转换的类型指定了在 `expression` 上执行某种特定类型的转换。

1. dynamic_cast

dynamic_cast 支持运行时识别指针或引用所指向的对象。对 dynamic_cast 的讨论将在 18.2 节中进行。

2. const_cast

const_cast, 顾名思义, 将转换掉表达式的 const 性质。例如, 假设有函数 string_copy, 我们对其唯一的 char* 类型的参数只读不写。在访问该函数时, 最好的选择是修改它让它接受 const char* 类型的参数。如果不行, 可通过 const_cast 用一个 const 值调用 string_copy 函数:

```
const char *pc_str;
char *pc = string_copy(const_cast<char*>(pc_str));
```

只有使用 const_cast 才能将 const 性质转换掉。在这种情况下, 试图使用其他三种形式的强制转换都会导致编译时的错误。类似地, 除了添加或删除 const 特性, 用 const_cast 符来执行其他任何类型转换, 都会引起编译错误。

184

3. static_cast

编译器隐式执行的任何类型转换都可以由 static_cast 显式完成:

```
double d = 97.0;
// cast specified to indicate that the conversion is intentional
char ch = static_cast<char>(d);
```

当需要将一个较大的算术类型赋值给较小的类型时, 使用强制转换非常有用。此时, 强制类型转换告诉程序的读者和编译器: 我们知道并且不关心潜在的精度损失。对于从一个较大的算术类型到一个较小类型的赋值, 编译器通常会产生警告。当我们显式地提供强制类型转换时, 警告信息就会被关闭。

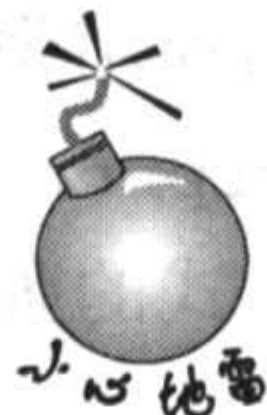
如果编译器不提供自动转换, 使用 static_cast 来执行类型转换也是很有用的。例如, 下面的程序使用 static_cast 找回存放在 void* 指针中的值 (4.2.2 节):

```
void* p = &d; // ok: address of any data object can be stored in a void*
// ok: converts void* back to the original pointer type
double *dp = static_cast<double*>(p);
```

可通过 static_cast 将存放在 void* 中的指针值强制转换为原来的指针类型, 此时我们应确保保持指针值。也就是说, 强制转换的结果应与原来的地址值相等。

4. reinterpret_cast

reinterpret_cast 通常为操作数的位模式提供较低层次的重新解释。



reinterpret_cast 本质上依赖于机器。为了安全地使用 reinterpret_cast, 要求程序员完全理解所涉及的数据类型, 以及编译器实现强制类型转换的细节。

例如, 对于下面的强制转换:

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

程序员必须永远记得 `pc` 所指向的真实对象其实是 `int` 型，而并非字符数组。任何假设 `pc` 是普通字符指针的应用，都有可能带来有趣的运行时错误。例如，下面语句用 `pc` 来初始化一个 `string` 对象：

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

它可能会引起运行时的怪异行为。

185

用 `pc` 初始化 `str` 这个例子很好地说明了显式强制转换是多么的危险。问题源于类型已经改变时编译器没有提供任何警告或错误提示。当我们用 `int` 型地址初始化 `pc` 时，由于显式地声明了这样的转换是正确的，因此编译器不提供任何错误或警告信息。后面对 `pc` 的使用都假设它存放的是 `char*` 型对象的地址，编译器确实无法知道 `pc` 实际上是指向 `int` 型对象的指针。因此用 `pc` 初始化 `str` 是完全正确的——虽然实际上是无意义的或是错误的。查找这类问题的原因相当困难，特别是如果 `ip` 到 `pc` 的强制转换和使用 `pc` 初始化 `string` 对象这两个应用发生在不同文件中的时候。

建议：避免使用强制类型转换

强制类型转换关闭或挂起了正常的类型检查（2.3 节）。强烈建议程序员避免使用强制类型转换，不依赖强制类型转换也能写出很好的 C++ 程序。

这个建议在如何看待 `reinterpret_cast` 的使用时非常重要。此类强制转换总是非常危险的。相似地，使用 `const_cast` 也总是预示着设计缺陷。设计合理的系统应不需要使用强制转换抛弃 `const` 特性。其他的强制转换，如 `static_cast` 和 `dynamic_cast`，各有各的用途，但都不应频繁使用。每次使用强制转换前，程序员应该仔细考虑是否还有其他不同的方法可以达到同一目的。如果非强制转换不可，则应限制强制转换值的作用域，并且记录所有假定涉及的类型，这样能减少错误发生的机会。

5.12.7 旧式强制类型转换

在引入命名的强制类型转换操作符之前，显式强制转换用圆括号将类型括起来实现：

```
char *pc = (char*) ip;
```

效果与使用 `reinterpret_cast` 符号相同，但这种强制转换的可视性比较差，难以跟踪错误的转换。

标准 C++ 为了加强类型转换的可视性，引入命名的强制转换操作符，为程序员在必须使用强制转换时提供了更好的工具。例如，非指针的 `static_cast` 和 `const_cast` 要比 `reinterpret_cast` 更安全。结果使程序员（以及读者和操纵程序的工具）可清楚地辨别代码中每个显式的强制转换潜在的风险级别。



最佳
实践

虽然标准 C++ 仍然支持旧式强制转换符号，但是我们建议，只有在 C 语言或标准 C++ 之前的编译器上编写代码时，才使用这种语法。

旧式强制转换符号有下列两种形式：

```
type (expr); // Function-style cast notation
```

```
(type) expr; // C-language-style cast notation
```

186

旧式强制转换依赖于所涉及的数据类型，具有与 `const_cast`、`static_cast` 和 `reinterpret_cast` 一样的行为。在合法使用 `static_cast` 或 `const_cast` 的地方，旧式强制转换提供了与各自对应的命名强制转换一样的功能。如果这两种强制转换均不合法，则旧式强制转换执行 `reinterpret_cast` 功能。例如，我们可用旧式符号重写上一节的强制转换：

```
int ival; double dval;
ival += int (dval); // static_cast: converts double to int
const char* pc_str;
string_copy((char*)pc_str); // const_cast: casts away const
int *ip;
char *pc = (char*)ip; // reinterpret_cast: treats int* as char*
```

支持旧式强制转换符号是为了对“在标准 C++ 之前编写的程序”保持向后兼容性，并保持与 C 语言的兼容性。

习题

习题 5.32 给定下列定义：

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

指出可能发生的（如果有的话）隐式类型转换：

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

习题 5.33 给定下列定义：

```
int ival; double dval;
const string *ps; char *pc; void *pv;
```

用命名的强制类型转换符号重写下列语句：

```
(a) pv = (void*)ps; (b) ival = int(*pc);
(c) pv = &dval; (d) pc = (char*) pv;
```

187

小结

C++ 提供了丰富的操作符，并定义了用于内置类型值时操作符的含义。除此之外，C++ 还支持操作符重载，允许由程序员自己来定义用于类类型时操作符的含义。我们将在第 14 章中学习如何重载自定义的操作符。

要理解复合表达式（即含有多个操作符的表达式）就必须先了解优先级、结合性以及操作数的求值次序。每一个操作符都有自己的优先级别和结合性。优先级规定复合表达式中操作符结合的方式，而结合性则决定同一个优先级的操作符如何结合。

大多数操作符没有规定其操作数的求值顺序：由编译器自由选择先计算左操作数还是右操作

数。通常，操作数的求值顺序不会影响表达式的结果。但是，如果操作符的两个操作数都与同一个对象相关，而且其中一个操作数改变了该对象的值，则程序将会因此而产生严重的错误——而且这类错误很难发现。

最后，可以使用某种类型编写表达式，而实际需要的是另一类型的值。此时，编译器自动实现类型转换（既可以是内置类型也可以是为类类型而定义的），将特定类型转换为所需的类型。C++还提供了强制类型转换显式地将数值转换为所需的数据类型。

术语

arithmetic conversion (算术转换) 算术类型之间的转换。在使用二元算术操作符的地方，算术转换通常将较小的类型转换为较大的类型，以确保精度（例如，将小的整型 char 型和 short 型转换为 int 型）。

associativity (结合性) 决定同一优先级的操作符如何结合。C++的操作符要么是左结合（操作符从左向右结合）要么是右结合（操作符从右向左结合）。

binary operator (二元操作符) 有两个操作数的操作符。

cast (强制类型转换) 显式的类型转换。

compound expression (复合表达式) 含有多个操作符的表达式。

const_cast 将 const 对象转换为相应的非 const 类型的强制转换。

conversion (类型转换) 将某种类型的值转换为另一种类型值的处理方式。C++语言定义了内置类型之间的类型转换，也允许将某种类型转换为类类型或将类类型转换为某种类型。

dangling pointer (悬垂指针) 指向曾经存在的对象的指针，但该对象已经不再存在了。悬垂指针容易导致程序错误，而且这种错误很难检测出来。

delete expression (delete 表达式) delete

表达式用于释放由 new 动态分配的内存。

delete 有两种语法形式：

```
delete p;           // delete object
delete [] p;        // delete array
```

第一种形式的 p 必须是指向动态创建对象的指针；第二种形式的 p 则应指向动态创建数组的第一个元素。C++程序使用 delete 取代 C 语言的标准库函数 free。

dynamic_cast 用于结合继承和运行时类型识别。参见 18.2 节。

expression (表达式) C++程序中的最低级的计算。表达式通常将一个操作符用于一个或多个操作数。每个表达式产生一个结果。表达式也可用作操作数，因此可用多个操作符编写复合表达式。

implicit conversion (隐式类型转换) 编译器自动实现的类型转换。假设表达式需要某种特定类型的数值，但其操作数却是其他不同的类型，此时如果系统定义了适当的类型转换，编译器会自动根据转换规则将该操作数转换为需要的类型。

integral promotion (整型提升) 整型提升是标准类型转换规则的子集，它将较小的整型转换为最接近的较大数据类型。整型（如 short、char 等）被提升为 int 型或 unsigned int 型。

new expression (new 表达式) new 表达式用于运行时从自由存储区中分配内存空间。本章

使用 `new` 创建单个对象，其语法形式为：

```
new type;
new type(inits);
```

`new` 表达式创建指定 `type` 类型的对象，并且可选择在创建时使用 `inits` 初值初始化该对象，然后返回指向该对象的指针。C++ 程序使用 `new` 取代 C 语言的标准库函数 `malloc`。

operand (操作数) 表达式操纵的值。

operator (操作符) 决定表达式执行什么功能的符号。C++ 语言定义了一组操作符以及将它们用于内置类型时的含义，还定义了每个操作符的优先级和结合性以及它们所需要的操作数个数。C++ 语言允许重载操作符，以使它们能用于类类型的对象。

operator overloading (操作符重载) 对操作符的功能重定义以用于类类型。我们将在第14章中学习如何重载不同的操作符版本。

order of evaluation (求值顺序) 操作符的操作数计算顺序（如果有的话）。大多数情况下，C++ 编译器可自由选择操作数求解的次序。

precedence (优先级) 定义了复合表达式中不同操作符的结合方式。高优先级的操作符要比低优先级操作符结合得更紧密。

reinterpret_cast 将操作数内容解释为另一种不同的类型。这类强制转换本质上依赖于机器，而且是非常危险的。

result (结果) 计算表达式所获得的值或对象。

static_cast 编译器隐式执行的任何类型转换都可以由 `static_cast` 显式完成。我们常常使用 `static_cast` 取代由编译器实现的隐式转换。

unary operator (一元操作符) 只有一个操作数的操作符。

~ operator (~操作符) 逐位求反操作符，将其操作数的每个位都取反。

, operator (逗号操作符) 逗号操作符。用逗号隔开的表达式从左向右计算，整个逗号表达式的结果为其最右边的表达式的值。

?: operator (?:操作符) 条件操作符。if-then-else 表达式的形式为：

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

如果条件 `cond` 为 `true`，则计算 `expr1`，否则计算 `expr2`。

& operator (&操作符) 位与操作符。在做位与操作时，如果两个操作数对应位置上的位都为 1，则操作结果中该位为 1，否则为 0，最后位与操作产生一个新的整数值。

^ operator (^操作符) 位异或操作符。在做位异或操作时，如果两个操作数对应位置上的位只有一个（注意不是两个）为 1，则操作结果中该位为 1，否则为 0，位异或操作产生一个新的整数值。

| operator (|操作符) 位或操作符。在做位或操作时，如果两个操作数对应位置上的位至少有一个为 1，则操作结果中该位为 1，否则为 0，位或操作产生一个新的整数值。

++ operator (++操作符) 自增操作符。自增操作符有两种形式：前置操作和后置操作。前置自增操作生成左值，在给操作数加 1 后返回改变后的操作数值。后置自增操作生成右值，给操作数加 1 但返回未改变的操作数原值。

-- operator (--操作符) 自减操作符。自减操作符也有两种形式：前置操作和后置操作。前置自减操作生成左值，在给操作数减 1 后返回改变后的操作数值。后置自减操作生成右值，给操作数减 1 但返回未改变的操作数原值。

<< operator (<<操作符) 左移操作符。左移

操作符将其左操作数的各个位向左移动若干个位，移动的位数由其右操作数指定。左移操作符的右操作数必须是 0 值或正整数，而且它的值必须严格小于左操作数的位数。

>> operator (>>操作符) 右移操作符。与左移操作符类似，右移操作符将其左操作数的各个位向右移动，其右操作数必须是 0 值或正整数，而且它的值必须严格小于左操作数的位数。

188
190

第 6 章

语 句

目录

6.1 简单语句.....	166
6.2 声明语句.....	167
6.3 复合语句（块）	167
6.4 语句作用域.....	168
6.5 if 语句.....	169
6.6 switch 语句	172
6.7 while 语句.....	177
6.8 for 循环语句.....	179
6.9 do while 语句.....	182
6.10 break 语句.....	183
6.11 continue 语句.....	184
6.12 goto 语句	185
6.13 try 块和异常处理.....	186
6.14 使用预处理器进行调试.....	190
小结.....	192
术语.....	192

语句类似于自然语言中的句子。C++语言既有只完成单一任务的简单语句，也有作为一个单元执行的由一组语句组成的复合语句。和大多数语言一样，C++也提供了实现条件分支结构的语句以及重复地执行同一段代码的循环结构。本章将详细讨论 C++所支持的语句。

通常情况下，语句是顺序执行的。但是，除了最简单的程序外，只有顺序执行往往并不足够。为此，C++定义了一组控制流（flow-of-control）语句，允许有条件地执行或者重复地执行某部分功能。if 和 switch 语句提供了条件分支结构，而 for、while 和 do while 语句则支持重复执行的功能。后几种语句常称为循环或者迭代语句。

6.1 简单语句

C++中，大多数语句以分号结束。例如，像 `ival+5` 这样的表达式，在后面加上分号，就是一条表达式语句（expression statement）。表达式语句用于计算表达式。但执行下面的语句

```
ival + 5;    // expression statement
```

却没有任何意义：因为计算出来的结果没有用于赋值或其他用途。通常，表达式语句所包含的表达式在计算时会影响程序的状态，使用赋值、自增、输入或输出操作符的表达式就是很好的例子。

空语句

程序语句最简单的形式是空语句（null statement），它使用以下的形式（只有一个单独的分号）：

```
;    // null statement
```

如果在程序的某个地方，语法上需要一个语句，但逻辑上并不需要，此时应该使用空语句。这种用法常见于在循环条件判断部分就能完成全部循环工作的情况。例如，下面程序从输入流中读取数据，在获得某个特殊值前无需作任何操作：

```
// read until we hit end-of-file or find an input equal to sought
while (cin >> s && s != sought)
    ;    // null statement
```

循环条件从标准输入中读入一个值并检验 cin 的读入是否成功。如果成功读取数据，循环条件紧接着检查该值是否等于 sought。如果找到了需要的值，则退出 while 循环；否则，循环条件再次从 cin 里读入另一个值继续检验。



最佳
实践

使用空语句时应该加上注释，以便任何读这段代码的人都知道该语句是有意省略的。

由于空语句也是一个语句，因此可用在任何允许使用语句的地方。由于这个原因，那些看似非法的分号往往只不过是一个空语句而已：

```
// ok: second semicolon is superfluous null statement
ival = v1 + v2;;
```

这个程序段由两条语句组成：一条表达式语句和一条空语句。



小心地雷

无关的空语句并非总是无害的。

在 while 或 if 条件后面额外添加分号，往往会彻底改变程序员的意图：

```
// disaster: extra semicolon: loop body is this null statement
while (iter != svec.end()) ; // null statement--while body is empty!
    ++iter;           // increment is not part of the loop
```

这个程序将会无限次循环。与缩进的意义相反，此自增语句并不是循环的一部分。由于循环条件后面多了一个分号，因此循环体为空语句。

6.2 声明语句

在 C++ 中，对象或类的定义或声明也是语句。尽管定义语句这种说法也许更准确些，但定义语句经常被称为**声明语句**（declaration statement）。2.3 节介绍了变量的定义和声明。2.8 节介绍了类的定义，相关内容将在第 12 章进一步探讨。

6.3 复合语句（块）

复合语句（compound statement），通常被称为**块**（block），是用一对花括号括起来的语句序列（也可能是空的）。块标识了一个作用域，在块中引入的名字只能在该块内部或嵌套在块中的子块里访问。通常，一个名字只从其定义处到该块的结尾这段范围内可见。

复合语句用在语法规则要求使用单个语句但程序逻辑却需要不止一个语句的地方。例如，while 或 for 语句的循环体必须是单个语句。然而，大多数情况都需要在循环体里执行多个语句。因而可使用一对花括号将语句序列括起来，使其成为块语句。

回顾一下 1.6 节处理书店问题的程序，以其中用到的 while 循环为例：

```
// if so, read the transaction records
while (std::cin >> trans)
    if (total.same_isbn(trans))
        // match: update the running total
        total = total + trans;
else {
    // no match: print & assign to total
    std::cout << total << std::endl;
    total = trans;
}
```

193

在 else 分支中，程序逻辑需要输出 total 的值，然后用 trans 重置 total。但是，else 分支只能后接单个语句。于是，用一对花括号将上述两条语句括起来，使其在语法上成为单个语句（复合语句）。这个语句既符合语法规则又满足程序的需要。



与其他大多数语句不同，块并不是以分号结束的。

像空语句一样，程序员也可以定义空块，用一对内部没有语句的花括号实现：

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

习题

- 习题 6.1 什么是空语句？请给出一个使用空语句的例子。
- 习题 6.2 什么是块语句？请给出一个使用块的例子。
- 习题 6.3 使用逗号操作符（5.9 节）重写书店问题中 while 循环里的 else 分支，使它不再需要用块实现。解释一下重写后是提高还是降低了该段代码的可读性。
- 习题 6.4 在解决书店问题的 while 循环中，如果删去 while 后面的左花括号及相应的右花括号，将会给程序带来什么影响？

6.4 语句作用域

有些语句允许在它们的控制结构中定义变量：

```
while (int i = get_num())
    cout << i << endl;
i = 0; // error: i is not accessible outside the loop
```

194

注解

在条件表达式中定义的变量必须初始化，该条件检验的就是初始化对象的值。

在语句的控制结构中定义的变量，仅在定义它们的块语句结束前有效。这种变量的作用域限制在语句体内。通常，语句体本身就是一个块语句，其中也可能包含了其他的块。一个在控制结构里引入的名字是该语句的局部变量，其作用域局限在语句内部。

```
// index is visible only within the for statement
for (vector<int>::size_type index = 0;
     index != vec.size(); ++index)
{ // new scope, nested within the scope of this for statement
    int square = 0;
    if (index % 2) // ok: index is in scope
        square = index * index;
    vec[index] = square;
}
if (index != vec.size()) // error: index is not visible here
```

如果程序需要访问某个控制结构中的变量，那么这个变量必须在控制语句外部定义。

```
vector<int>::size_type index = 0;
for ( /* empty */ ; index != vec.size(); ++index)
    // as before
if (index != vec.size()) // ok: now index is in scope
    // as before
```

注解

早期的 C++ 版本以不同的方式处理 for 语句中定义的变量的作用域：将 for 语句头定义的变量视为在 for 语句之前定义。有些更旧式的 C++ 程序代码允许在 for 语句作用域外访问控制变量。

对于在控制语句中定义的变量，限制其作用域的一个好处是，这些变量名可以重复使用而不

必担心它们的当前值在每一次使用时是否正确。对于作用域外的变量，是不可能用到其在作用域内的残留值的。

6.5 if 语句

if 语句根据特定表达式是否为真来有条件地执行另一个语句。if 语句有两种形式：其中一种带 **else** 分支而另一种则没有。根据语法结构，最简单的 if 语句是这样的：

```
if (condition)
    statement
```

其中的 *condition* 部分必须用圆括号括起来。它可以是一个表达式，例如：

```
if (a + b > c) { /* ... */ }
```

或者一个初始化声明，例如：

```
// ival only accessible within the if statement
if (int ival = compute_value()) { /* ... */ }
```

通常，*statement* 部分可以是复合语句，即用花括号括起来的块语句。

如果在条件表达式中定义了变量，那么变量必须初始化。将已初始化的变量值转换为 **bool** 值（5.12.3 节）后，该 **bool** 值决定条件是否成立。变量类型可以是任何可转换为 **bool** 型的类型，这意味着它可以是算术类型或指针类型。正如第 14 章要提及的，一个类类型能否用在条件表达式中取决于类本身。迄今为止，在所有用过的类类型中，**IO** 类型可以用作条件，但 **vector** 类型和 **string** 类型一般不可用作条件。

为了说明 if 语句的用法，下面程序用于寻找 **vector<int>** 对象中的最小值，并且记录这个最小值出现的次数。为了解决这个问题，需要两个 if 语句：一个判断是否得到一个新的最小值，而另一个则用来增加当前最小值的数目。

```
if (minVal > ivec[i]) { /* process new minVal */ }
if (minVal == ivec[i]) { /* increment occurrence count */ }
```

语句块用作 if 语句的对象

现在单独考虑上述例子中的每个 if 语句。其中一个 if 语句将要决定是否出现了一个新的最小值，如果是的话，则要重置计数器并更新最小值：

```
if (minVal > ivec[i]) { // execute both statements if condition is true
    minVal = ivec[i];
    occurs = 1;
}
```

另一个 if 语句则有条件地更新计数器，它只需要一个语句，因此不必用花括号括起来：

```
if (minVal == ivec[i])
    ++occurs;
```



当多个语句必须作为单个语句执行时，比较常见的错误是漏掉了花括号。

在下面的程序中，与程序员缩进目的相反，对 `occurs` 的赋值并不是 `if` 语句的一部分：

```
// error: missing curly brackets to make a block!
if (minVal > ivec[i])
    minVal = ivec[i];
    occurs = 1; // executed unconditionally: not part of the if
```

196

这样写的话，对 `occurs` 的赋值将会无条件地执行。这种错误很难发现，因为程序代码看起来是正确的。



最佳
实践

很多编辑器和开发环境都提供工具自动根据语句结构缩排源代码。有效地利用这些工具将是一种很好的编程方法。

if 语句的 else 分支

紧接着，我们要考虑如何将那些 `if` 语句放在一起形成一个执行语句序列。这些 `if` 语句的排列顺序非常重要。如果采用下面的顺序：

```
if (minVal > ivec[i]) {
    minVal = ivec[i];
    occurs = 1;
}
// potential error if minVal has just been set to ivec[i]
if (minVal == ivec[i])
    ++occurs;
```

那么计数器将永远得不到 1。这段代码只是对第一次出现的最小值重复计数。

这样两个 `if` 语句不但在值相同时执行起来有潜在的危险，而且还是没必要的。同一个元素不可能既小于 `minVal` 又等于它。如果其中一个条件是真的，那么另一个条件就可以安全地忽略掉。`if` 语句为这种只能二选一的（either-or）条件提供了 `else` 子句。

if else 语句的语法形式为：

```
if (condition)
    statement1
else
    statement2
```

如果 `condition` 为真，则执行 `statement1`；否则，执行 `statement2`：

```
if (minVal == ivec[i])
    ++occurs;
else if (minVal > ivec[i]) {
    minVal = ivec[i];
    occurs = 1;
}
```

值得注意的是，`statement2` 既可以是任意语句，也可以是用花括号括起来的块语句。在这个例子里，`statement2` 本身是一个 `if` 语句。

悬垂 else

对于 `if` 语句的使用，还有一个重要的复杂问题没有考虑。上述例子中，注意到没有一个 `if` 分支能直接处理元素值大于 `minVal` 的情况。从逻辑上来说，可以忽略这些元素——如果该元素

197

比当前已找到的最小值大，那就应该没什么要做的。然而，通常需要使用 if 语句为三种不同情况提供执行的内容，即如果一个值大于、小于或等于其他值时，可能都需要执行特定的步骤。为此重写循环部分，显式地处理这三种情况：

```
// note: indented to make clear how the else branches align with the corresponding if
if (minVal < ivec[i])
    { } // empty block
else if (minVal == ivec[i])
    ++occurs;
else { // minVal > ivec[i]
    minVal = ivec[i];
    occurs = 1;
}
```

上述的三路测试精确地控制了所有的情况。然而，简单地把前两个情况用一个嵌套 if 语句实现将会产生问题：

```
// oops: incorrect rewrite: This code won't work!
if (minVal <= ivec[i])
    if (minVal == ivec[i])
        ++occurs;
else { // this else goes with the inner if, not the outer one!
    minVal = ivec[i];
    occurs = 1;
}
```



这个版本表明了所有语言的 if 语句都普遍存在着潜在的二义性。这种情况往往称为悬垂 else (dangling-else) 问题，产生于一个语句包含的 if 子句多于 else 子句时：对于每一个 else，究竟它们归属哪个 if 语句？

在上述的代码中，缩进的用法表明 else 应该与外层的 if 子句匹配。然而，C++ 中悬垂 else 问题带来的二义性，通过将 else 匹配给最后出现的尚未匹配的 if 子句来解决。在这个情况下，这个 if else 语句实际上等价于下面的程序：

```
// oops: still wrong, but now the indentation matches execution path
if (minVal <= ivec[i])
    // indented to match handling of dangling-else
    if (minVal == ivec[i])
        ++occurs;
else {
    minVal = ivec[i];
    occurs = 1;
}
```

可以通过用花括号将内层的 if 语句括起来成为复合语句，从而迫使这个 else 子句与外层的 if 匹配。

```
if (minVal <= ivec[i]) {
    if (minVal == ivec[i])
        ++occurs;
} else {
    minVal = ivec[i];
}
```

```
occurs = 1;
```

提示

有些编程风格建议总是在 if 后面使用花括号。这样做可以避免日后修改代码时产生混乱和错误。至少，无论 if (或者 while) 后面是简单语句，例如赋值和输出语句，还是其他任意语句，使用花括号都是一个比较好的做法。

习题

习题 6.5 改正下列代码：

```
(a) if (ival1 != ival2)
    ival1 = ival2
    else ival1 = ival2 = 0;

(b) if (ival < minval)
    minval = ival; // remember new minimum
    occurs = 1;    // reset occurrence counter

(c) if (int ival = get_value())
    cout << "ival = " << ival << endl;
    if (!ival)
    cout << "ival = 0\n";

(d) if (ival = 0)
    ival = get_value();
```

习题 6.6 什么是“悬垂 else”？C++是如何匹配 else 子句的？

6.6 switch 语句

深层嵌套的 if-else 语句往往在语法上是正确的，但逻辑上却没有正确地反映程序员的意图。例如，错误的 else-if 匹配很容易被忽略。添加新的条件和逻辑关系，或者对语句做其他修改，都很难保证正确。**switch** 语句提供了一种更方便的方法来实现深层嵌套的 if/else 逻辑。

199 假设要统计五个元音在文本里分别出现的次数，程序逻辑结构如下：

- 按顺序读入每个字符直到读入完成为止。
- 把每个字符与元音字符集做比较。
- 如果该字符与某个元音匹配，则该元音的计数器加 1。
- 显示结果。

使用此程序分析本章的英文版，得到以下输出结果：

```
Number of vowel a: 3499
Number of vowel e: 7132
Number of vowel i: 3577
Number of vowel o: 3530
Number of vowel u: 1185
```

6.6.1 使用 switch

直接使用 switch 语句解决上述问题:

```
char ch;
// initialize counters for each vowel
int aCnt = 0, eCnt = 0, iCnt = 0,
    oCnt = 0, uCnt = 0;
while (cin >> ch) {
    // if ch is a vowel, increment the appropriate counter
    switch (ch) {
        case 'a':
            ++aCnt;
            break;
        case 'e':
            ++eCnt;
            break;
        case 'i':
            ++iCnt;
            break;
        case 'o':
            ++oCnt;
            break;
        case 'u':
            ++uCnt;
            break;
    }
}
// print results
cout << "Number of vowel a: \t" << aCnt << '\n'
    << "Number of vowel e: \t" << eCnt << '\n'
    << "Number of vowel i: \t" << iCnt << '\n'
    << "Number of vowel o: \t" << oCnt << '\n'
    << "Number of vowel u: \t" << uCnt << endl;
```

通过对圆括号内表达式的值与其后列出的关键字做比较, 实现 switch 语句的功能。表达式必须产生一个整数结果, 其值与每个 case 的值比较。关键字 case 和它所关联的值称为 **case 标号** (case label)。每个 case 标号的值都必须是一个常量表达式 (2.7 节)。除此之外, 还有一个特殊的 case 标号——default 标号, 我们将在 6.6.3 节介绍它。

200

如果表达式与其中一个 case 标号的值匹配, 则程序将从该标号后面的第一个语句开始依次执行各个语句, 直到 switch 结束或遇到 break 语句为止。如果没有发现匹配的 case 标号 (并且也没有 default 标号), 则程序从 switch 语句后面的第一条语句继续执行。在这个程序中, switch 语句是 while 循环体中唯一的语句, 于是, switch 语句匹配失败后, 将控制流返回给 while 循环条件。

6.10 节将讨论 break 语句。简单地说, break 语句中断当前的控制流。对于 switch 的应用, break 语句将控制跳出 switch, 继续执行 switch 语句后面的第一个语句。在这个例子中, 正如大家所知道的, 将会把控制转移到 switch 后面的下一语句, 即交回给 while。

6.6.2 switch 中的控制流

了解 case 标号的执行流是必要的。



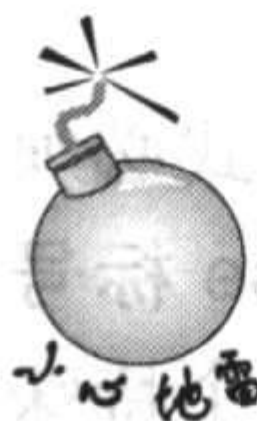
存在一个普遍的误解：以为程序只会执行匹配的 case 标号相关联的语句。实际上，程序从该点开始执行，并跨越 case 边界继续执行其他语句，直到 switch 结束或遇到 break 语句为止。

有时候，这种行为的确是正确的。程序员也许希望执行完某个特定标号的代码后，接着执行后续标号关联的语句。但更常见的是，我们只需要执行某个特定标号对应的代码。为了避免继续执行其后续 case 标号的内容，程序员必须利用 break 语句清楚地告诉编译器停止执行 switch 中的语句。大多数情况下，在下一个 case 标号之前的最后一条语句是 break。例如，下面统计元音出现次数的 switch 语句是不正确的：

```
// warning: deliberately incorrect!
switch (ch) {
    case 'a':
        ++aCnt;    // oops: should have a break statement
    case 'e':
        ++eCnt;    // oops: should have a break statement
    case 'i':
        ++iCnt;    // oops: should have a break statement
    case 'o':
        ++oCnt;    // oops: should have a break statement
    case 'u':
        ++uCnt;    // oops: should have a break statement
}
```

201

为了搞清楚该程序导致了什么结果，假设 ch 的值是 'i' 来跟踪这个版本的代码。程序从 case 'i' 后面的语句开始执行，iCnt 的值加 1。但是，程序的执行并没有在这里停止，而是越过 case 标号继续执行，同时将 oCnt 和 uCnt 的值都加了 1。如果 ch 是 'e' 的话，那么 eCnt、iCnt、oCnt 以及 uCnt 的值都会加 1。



对于 switch 结构，漏写 break 语句是常见的程序错误。



尽管没有严格要求在 switch 结构的最后一个标号之后指定 break 语句，但是，为了安全起见，最好在每个标号后面提供一个 break 语句，即使是最后一个标号也一样。如果以后在 switch 结构的末尾又需要添加一个新的 case 标号，则不用再在前面加 break 语句了。

慎用 break 语句，它并不总是恰当的

有这么一种常见的情况，程序员希望在 case 标号后省略 break 语句，允许程序向下执行多个 case 标号。这时，两个或多个 case 值由相同的动作序列来处理。由于系统限定一个 case 标号只能与一个值相关联，于是为了指定一个范围，典型的做法是，把 case 标号依次排列。例如，如果只希望计算文本中元音的总数，而不是每一个元音的个数，则可以这样写：

```
int vowelCnt = 0;
// ...
switch (ch)
{
```

```

// any occurrence of a,e,i,o,u increments vowelCnt
case 'a':
case 'e':
case 'i':
case 'o':
case 'u':
    ++vowelCnt;
    break;
}

```

每个 case 标号不一定要另起一行。为了强调这些 case 标号表示的是一个要匹配的范围，可以将它们全部在一行中列出：

```

switch (ch)
{
    // alternative legal syntax
    case 'a': case 'e': case 'i': case 'o': case 'u':
        ++vowelCnt;
        break;
}

```

比较少见的用法是，为了执行某个 case 的代码后继续执行下一个 case 的代码，故意省略 break 语句。



最佳
实践

故意省略 case 后面的 break 语句是很罕见的，因此应该提供一些注释说明其逻辑。

202

6.6.3 default 标号

default 标号 (default label) 提供了相当于 else 子句的功能。如果所有的 case 标号与 switch 表达式的值都不匹配，并且 default 标号存在，则执行 default 标号后面的语句。例如，在上述例子中添加一个计数器 otherCnt 统计读入多少个辅音字母，为 switch 结构增加 default 标号，其标志的分支实现 otherCnt 的自增：

```

// if ch is a vowel, increment the appropriate counter
switch (ch) {
    case 'a':
        ++aCnt;
        break;
    // remaining vowel cases as before
    default:
        ++otherCnt;
        break;
}
}

```

在这个版本的代码中，如果 ch 不是元音，程序流程将执行 default 标号的相关语句，使 otherCnt 的值加 1。



最佳
实践

哪怕没有语句要在 default 标号下执行，定义 default 标号仍然是有用的。定义 default 标号是为了告诉它的读者，表明这种情况已经考虑到了，只是没什么要执行的。

一个标号不能独立存在，它必须位于语句之前。如果 switch 结构以 default 标号结束，而

且 default 分支不需要完成任何任务，那么该标号后面必须有一个空语句。

6.6.4 switch 表达式与 case 标号

switch 求解的表达式可以非常复杂。特别是，该表达式也可以定义和初始化一个变量：

```
switch(int ival = get_response())
```

在这个例子中，ival 被初始化为 get_response 函数的调用结果，其值将要与每个 case 标号作比较。变量 ival 始终存在于整个 switch 语句中，在 switch 结构外面该变量就不再有效了。

case 标号必须是整型常量表达式（2.7 节）。例如，下面的标号将导致编译时的错误：

```
// illegal case label values
case 3.14: // noninteger
case ival: // nonconstant
```

如果两个 case 标号具有相同的值，同样也会导致编译时的错误。

6.6.5 switch 内部的变量定义

对于 switch 结构，只能在它的最后一个 case 标号或 default 标号后面定义变量：

```
case true:
    // error: declaration precedes a case label
    string file_name = get_file_name();
    break;
case false:
    // ...
```

制定这个规则是为了避免出现代码跳过变量的定义和初始化的情况。

回顾变量的作用域，变量从它的定义点开始有效，直到它所在块结束为止。现在考虑如果在两个 case 标号之间定义变量会出现什么情况。该变量会在块结束之前一直存在。对于定义该变量的标号后面的其他 case 标号，它们所关联的代码都可以使用这个变量。如果 switch 从那些后续 case 标号开始执行，那么这个变量可能还未定义就要使用了。

在这种情况下，如果需要为某个特殊的 case 定义变量，则可以引入块语句，在该块语句中定义变量，从而保证这个变量在使用前被定义和初始化。

```
case true:
{
    // ok: declaration statement within a statement block
    string file_name = get_file_name();
    // ...
}
break;
case false:
    // ...
```

习题

习题 6.7 前面已实现的统计元音的程序存在一个问题：不能统计大写的元音字母。编写程序统计大小写的元音，也就是说，你的程序计算出来的 aCnt，既包括 'a' 也包括 'A' 出现的次数，

其他四个元音也一样。

习题 6.8 修改元音统计程序使其可统计出读入的空格、制表符和换行符的个数。

习题 6.9 修改元音统计程序使其可统计以下双字符序列出现的次数：ff、fl 以及 fi。

习题 6.10 下面每段代码都暴露了一个常见编程错误。请指出并修改之。

```
(a) switch (ival) {
    case 'a': aCnt++;
    case 'e': eCnt++;
    default: iouCnt++;
}

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

(c) switch (ival) {
    case 1, 3, 5, 7, 9:
        oddcnt++;
        break;
    case 2, 4, 6, 8, 10:
        evencnt++;
        break;
}

(d) int ival=512 jval=1024, kval=4096;
    int bufsize;
    // ...
    switch(swt) {
        case ival:
            bufsize = ival * sizeof(int);
            break;
        case jval:
            bufsize = jval * sizeof(int);
            break;
        case kval:
            bufsize = kval * sizeof(int);
            break;
    }
```

6.7 while 语句

当条件为真 (true) 时，**while** 语句反复执行目标语句。它的语法形式如下：

```
while (condition)
    statement
```

只要条件 *condition* 的值为 *true*, 执行语句 *statement* (通常是一个块语句)。 *condition* 不能为空。如果第一次求解 *condition* 就产生 *false* 值, 则不执行 *statement*。

循环条件 *condition* 可以是一个表达式, 或者是提供初始化的变量定义。

```
bool quit = false;
while (!quit) { // expression as condition
    quit = do_something();
}
while (int loc = search(name)) { // initialized variable as condition
    // do something
}
```

在循环条件中定义的任意变量都只在与 *while* 关联的块语句中可见。每一次循环都将该变量的初值转换为 *bool* 类型 (5.12.3 节)。如果求得的值为 *true*, 则执行 *while* 的循环体。通常, 循环条件自身或者在循环体内必须做一些相关操作来改变循环条件表达式的值。否则, 循环可能永远不会结束。



在循环条件中定义的变量在每次循环里都要经历创建和撤销的过程。

while 循环的使用

前面的章节已经用过很多 *while* 循环, 但为了更完整地了解该结构, 考虑下面将一个数组的内容复制到另一个数组的例子:

```
// arr1 is an array of ints
int *source = arr1;
size_t sz = sizeof(arr1)/sizeof(*arr1); // number of elements
int *dest = new int[sz]; // uninitialized elements
while (source != arr1 + sz)
    *dest++ = *source++; // copy element and increment pointers
```

首先初始化 *source* 和 *dest*, 并使它们各自指向所关联的数组的第一个元素。*while* 循环条件判断是否已经到达要复制的数组的末尾。如果没有, 继续执行循环。循环体只有单个语句, 实现元素的复制, 并对两个指针做自增操作, 使它们指向对应数组的下一个元素。

正如 5.5 节提出的关于“简洁即是美”的建议, C++ 程序员应尝试编写简洁的表达式。*while* 循环体中的语句:

```
*dest++ = *source++;
```

是一个经典的例子。这个表达式等价于:

```
{
    *dest = *source; // copy element
    ++dest; // increment the pointers
    ++source;
}
```



提示 *while* 循环内的赋值操作是一种常见的用法。因为这类代码广为流传, 所以学习这种表达式非常重要, 要一眼就能看出其含义来。

习题

习题 6.11 解释下面的循环，更正你发现的问题。

```
(a) string bufString, word;
    while (cin >> bufString >> word) { /* ... */ }

(b) while (vector<int>::iterator iter != ivec.end())
    { /*... */ }

(c) while (ptr = 0)
    ptr = find_a_value();

(d) while (bool status = find(word))
    { word = get_next_word(); }
    if (!status)
        cout << "Did not find any words\n";
```

习题 6.12 编写一个小程序，从标准输入读入一系列 `string` 对象，寻找连续重复出现的单词。程序应该找出满足以下条件的单词的输入位置：该单词的后面紧跟着再次出现自己本身。跟踪重复次数最多的单词及其重复次数。输出重复次数的最大值，若没有单词重复则输出说明信息。例如，如果输入是：

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

则输出应表明“now”这个单词出现了三次。

习题 6.13 详细解释下面 `while` 循环中的语句是如何执行的：

```
*dest++ = *source++;
```

6.8 for 循环语句

`for` 语句的语法形式是：

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

init-statement（初始化语句）必须是声明语句、表达式语句或空语句。这些语句都以分号结束，因此其语法形式也可以看成：

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

当然，从技术上说，在 *initializer* 后面的分号是 `for` 语句头的一部分。

一般来说，*init-statement* 用于对每次循环过程中都要修改的变量进行初始化，或者赋给一个起始值。而 *condition*（循环条件）则是用来控制循环的。当 *condition* 为 `true` 时，循环执行 *statement*（循环体语句）。如果第一次求解 *condition* 就得 `false` 值，则不执行 *statement*。*expression* 通常用于修改在 *init-statement* 中初始化并在 *condition* 中检查的变量。它在每次循环迭代后都要求解。如果第一次求解 *condition* 就得 `false` 值，则始终不执行 *expression*。通常，*statement* 既可以是单个语句也可以是复合语句。

for 循环的使用

假设有下面的 for 循环，用于输出一个 vector 对象的内容：

```

for (vector<string>::size_type ind = 0;
     ind != svec.size(); ++ind) {
    cout << svec[ind]; // print current element
    // if not the last element, print a space to separate from the next one
    if (ind + 1 != svec.size())
        cout << " ";
}

```

它的计算顺序如下：

(1) 循环开始时，执行一次 *init-statement*（初始化语句）。在这个例子中，定义了 *ind*，并将它初始化为 0。

(2) 接着，求解 *condition*（循环条件）。如果 *ind* 不等于 *svec.size()*，则执行 for 循环体。否则，循环结束。如果在第一次循环时，条件就为 *false*，则不执行 for 循环体。

(3) 如果条件为 *true*，则执行 for 循环体。本例中，for 循环体输出当前元素值，并检验这个元素是否是最后一个。如果不是，则输出一个空格，用于分隔当前元素和下一个元素。

(4) 最后，求解 *expression*。本例中，*ind* 自增 1。

这四步描述了 for 循环的第一次完整迭代。接着重复第 2 步，然后是第 3、4 步，直到 *condition* 的值为 *false*，即 *ind* 等于 *svec.size()* 为止。



应该谨记：在 for 语句头定义的任何对象只限制在 for 循环体里可见。因此，对本例而言，在执行完 for 语句后，*ind* 不再有效（即不可访问）。

6.8.1 省略 for 语句头的某些部分

for 语句头中，可以省略 *init-statement*（初始化语句）、*condition*（循环条件）或者 *expression*（表达式）中的任何一个（或全部）。

如果不需要初始化或者初始化已经在别处实现了，则可以省略 *init-statement*。例如，使用迭代器代替下标重写输出 vector 对象内容的程序，为了提高易读性，可将初始化移到循环外面：

```

vector<string>::iterator iter = svec.begin();
for( /* null */ ; iter != svec.end(); ++iter) {
    cout << *iter; // print current element
    // if not the last element, print a space to separate from the next one
    if (iter+1 != svec.end())
        cout << " ";
}

```

注意此时必须要有一个分号表明省略了 *init-statement* —— 更准确地说，分号代表一个空的 *init-statement*。

省略 *condition*（循环条件），则等效于循环条件永远为 *true*：

```

for (int i = 0; /* no condition */ ; ++i)

```

相当于程序写为：

```
for (int i = 0; true ; ++i)
```

这么一来，循环体内就必须包含一个 `break` 或者 `return` 语句。否则，循环会一直执行直到耗尽系统的资源为止。同样地，如果省略 *expression*，则必须利用 `break` 或 `return` 语句跳出循环，或者在循环体内安排语句修改 *condition* 所检查的变量值。

```
for (int i = 0; i != 10; /* no expression */ ) {
    // body must change i or the loop won't terminate
}
```

如果循环体不修改 *i* 的值，则 *i* 始终为 0，循环条件永远成立。

209

6.8.2 for 语句头中的多个定义

可以在 `for` 语句的 *init-statement* 中定义多个对象；但是不管怎么样，该处只能出现一个语句，因此所有的对象必须具有相同的一般类型：

```
const int size = 42;
int val = 0, ia[size];
// declare 3 variables local to the for loop:
// ival is an int, pi a pointer to int, and ri a reference to int
for (int ival = 0, *pi = ia, &ri = val;
     ival != size;
     ++ival, ++pi, ++ri)
    // ...
```

习题

习题 6.14 解释下面每个循环，更正你发现的任何问题。

- (a)

```
for (int *ptr = &ia, ix = 0;
     ix != size && ptr != ia+size;
     ++ix, ++ptr) { /* ... */ }
```
- (b)

```
for (; ; ) {
    if (some_condition) return;
    // ...
}
```
- (c)

```
for (int ix = 0; ix != sz; ++ix) { /* ... */ }
if (ix != sz)
    // ...
```
- (d)

```
int ix;
for (ix != sz; ++ix) { /* ... */ }
```
- (e)

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

习题 6.15 `while` 循环特别擅长在某个条件保持为真时反复地执行；例如，当未到达文件尾时，一直读取下一个值。一般认为 `for` 循环是一种按步骤执行的循环：使用下标依次遍历集合中一定范围内的元素。按每种循环的习惯用法编写程序，然后再用另外一种结构重写。如果只能用一种循环来编写程序，你会选择哪种结构？为什么？

习题 6.16 给出两个 `int` 型的 `vector` 对象，编写程序判断一个对象是否是另一个对象的前缀。如果两个 `vector` 对象的长度不相同，假设较短的 `vector` 对象长度为 *n*，则只对这两个对象的前面 *n* 个元素做比较。例如，对于 `(0, 1, 1, 2)` 和 `(0, 1, 1, 2, 3, 5, 8)` 这两个 `vector`，你的程序应该返回 `true`。

6.9 do while 语句

210

在实际应用中，可能会要求程序员编写一个交互程序，为用户实现某种计算。一个简单的例子是：程序提示用户输入两个数，然后计算两者之和。在输出和值后，程序可以让用户选择是否重复这个过程计算下一个和。

程序的实现相当简单。只需输出一个提示，接着读入两个数，然后输出读入数之和。输出结果后，询问用户是否继续。

关键在于控制结构的选择。问题是要到用户要求退出时，才中止循环的执行。尤其是，在第一次循环时就要求一次和。do while 循环正好满足这样的需要。它保证循环体至少执行一次。其语句形式如下：

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



与 while 语句不同，do while 语句总是以分号结束。

在求解 *condition* 之前，先执行了 do 里面的 *statement*。*condition* 不能为空。如果 *condition* 的值为假，则循环结束，否则循环重复执行。使用 do while 循环，可以编写程序如下：

```
// repeatedly ask user for pair of numbers to sum
string rsp; // used in the condition; can't be defined inside the do
do {
    cout << "please enter two values: ";
    int val1, val2;
    cin >> val1 >> val2;
    cout << "The sum of " << val1 << " and " << val2
        << " = " << val1 + val2 << "\n\n";
    << "More? [yes][no] ";
    cin >> rsp;
} while (!rsp.empty() && rsp[0] != 'n');
```

循环体与之前编写的其他循环语句相似，因此很容易理解。奇怪的是此代码把 *rsp* 定义在 do 之前而不是在循环体内部。如果把 *rsp* 定义在 do 内部，那么 *rsp* 的作用域就被限制在 while 前的右花括号之前了。任何在循环条件中引用的变量都必须在 do 语句之前就已经存在。

因为要到循环语句或者语句块执行之后，才求解循环条件，因此 do while 循环不可以采用如下方式定义变量：

```
// error: declaration statement within do condition is not supported
do {
    // ...
    mumble(foo);
} while (int foo = get_foo()); // error: declaration in do condition
```

211

如果可以在循环条件中定义变量的话，则对变量的任何使用都将发生在变量定义之前！

习题

习题 6.17 解释下列的循环。更正你发现的问题。

```
(a) do
    int v1, v2;
    cout << "Please enter two numbers to sum:" ;
    cin >> v1 >> v2;
    if (cin)
        cout << "Sum is: "
            << v1 + v2 << endl;
    while (cin);

(b) do {
    // ...
} while (int ival = get_response());

(c) do {
    int ival = get_response();
    if (ival == some_value())
        break;
} while (ival);
if (!ival)
    // ...
```

习题 6.18 编写一个小程序，由用户输入两个 string 对象，然后报告哪个 string 对象按字母排列次序而言比较小（也就是说，哪个的字典序靠前）。继续要求用户输入，直到用户请求退出为止。请使用 string 类型、string 类型的小于操作符以及 do while 循环实现。

6.10 break 语句

break 语句用于结束最近的 while、do while、for 或 switch 语句，并将程序的执行权传递给紧接在被终止语句之后的语句。例如，下面的循环在 vector 中搜索某个特殊值的第一次出现。一旦找到，则退出循环：

```
vector<int>::iterator iter = vec.begin();
while (iter != vec.end()) {
    if (value == *iter)
        break; // ok: found it!
    else
        ++iter; // not found: keep looking
} // end of while
if (iter != vec.end()) // break to here ...
    // continue processing
```

本例中，break 终止了 while 循环。执行权交给紧跟在 while 语句后面的 if 语句，程序继续执行。

212

break 只能出现在循环或 switch 结构中，或者出现在嵌套于循环或 switch 结构中的语句里。对于 if 语句，只有当它嵌套在 switch 或循环里面时，才能使用 break。break 出现在循环外或者 switch 外将会导致编译时错误。当 break 出现在嵌套的 switch 或者循环语句中时，将会终止里层的 switch 或循环语句，而外层的 switch 或者循环不受影响：

```
string inBuf;
```

```

while (cin >> inBuf && !inBuf.empty()) {
    switch(inBuf[0]) {
        case '-':
            // process up to the first blank
            for (string::size_type ix = 1;
                ix != inBuf.size(); ++ix) {
                if (inBuf[ix] == ' ')
                    break; // #1, leaves the for loop
                // ...
            }
            // remaining '-' processing: break #1 transfers control here
            break; // #2, leaves the switch statement
        case '+':
            // ...
    } // end switch
    // end of switch: break #2 transfers control here
} // end while

```

#1 标记的 `break` 终止了连字符 ('-') case 标号内的 `for` 循环，但并没有终止外层的 `switch` 语句，而且事实上也并没有结束当前 case 语句的执行。接着程序继续执行 `for` 语句后面的第一个语句，即处理连字符 case 标号下的其他代码，或者执行结束这个 case 的 `break` 语句。

#2 标记的 `break` 终止了处理连字符情况的 `switch` 语句，但没有终止 `while` 循环。程序接着执行 `break` 后面的语句，即求解 `while` 的循环条件，从标准输入读入下一个 `string` 对象。

习题

习题 6.19 本节的第一个程序可以写得更简洁。事实上，该程序的所有工作可以全部包含在 `while` 的循环条件中。重写这个循环，使得它的循环体为空，并找出满足条件的元素。

习题 6.20 编写程序从标准输入读入一系列 `string` 对象，直到同一个单词连续出现两次，或者所有的单词都已读完，才结束读取。请使用 `while` 循环，每次循环读入一个单词。如果连续出现相同的单词，便以 `break` 语句结束循环，此时，请输出这个重复出现的单词；否则输出没有任何单词连续重复出现的信息。

213

6.11 continue 语句

continue 语句导致最近的循环语句的当次迭代提前结束。对于 `while` 和 `do while` 语句，继续求解循环条件。而对于 `for` 循环，程序流程接着求解 `for` 语句头中的 *expression* 表达式。

例如，下面的循环每次从标准输入中读入一个单词，只有以下划线开头的单词才做处理。如果是其他的值，终止当前循环，接着读取下一个单词：

```

string inBuf;
while (cin >> inBuf && !inBuf.empty()) {
    if (inBuf[0] != '_')
        continue; // get another input
    // still here? process string ...
}

```

`continue` 语句只能出现在 `for`、`while` 或者 `do while` 循环中，包括嵌套在这些循环内部的块语句中。