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

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

头文件不应包含 using 声明

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

一点注意事项

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

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



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

3.1 节练习

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

3.2 标准库类型 string



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

84

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

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



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

3.2.1 定义和初始化 string 对象

如何初始化类的对象是由类本身决定的。一个类可以定义很多种初始化对象的方式，只不过这些方式之间必须有所区别：或者是初始值的数量不同，或者是初始值的类型不同。表 3.1 列出了初始化 string 对象最常用的一些方式，下面是几个例子：

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

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

表 3.1：初始化 string 对象的方式

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

直接初始化和拷贝初始化

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

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

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

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

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

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

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

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

3.2.2 string 对象上的操作



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

表 3.2： string 的操作

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

读写 string 对象

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

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

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

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

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

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

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

读取未知数量的 string 对象

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

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

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

使用 getline 读取一整行

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

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

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

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



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

string 的 empty 和 size 操作

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

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

通过改写之前的程序，可以做到只输出非空的行：

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

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

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

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

string::size_type 类型

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

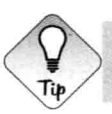
string 类及其他大多数标准库类型都定义了几种配套的类型。这些配套类型体现了标准库类型与机器无关的特性，类型 **size_type** 即是其中的一种。在具体使用的时候，通过作用域操作符来表明名字 size_type 是在类 string 中定义的。

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

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

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

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



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

比较 string 对象

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

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

相等性运算符（==和!=）分别检验两个 string 对象相等或不相等，string 对象相等意味着它们的长度相同而且所包含的字符也全都相同。关系运算符<、<=、>、>=分别检验一个 string 对象是否小于、小于等于、大于、大于等于另外一个 string 对象。上述这些运算符都依照（大小写敏感的）字典顺序：

89

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

下面是 string 对象比较的一个示例：

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

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

为 string 对象赋值

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

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

两个 string 对象相加

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

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

字面值和 string 对象相加

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

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

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

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

90

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

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

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

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

另一方面，s7 的初始化是非法的，根据其语义加上括号后就成了下面的形式：

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

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



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

3.2.2 节练习

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

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

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

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

3.2.3 处理 string 对象中的字符



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

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

91

下来。以往的经验告诉我们，处理这些情况常常要涉及语言和库的很多方面。

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

表 3.3：cctype 头文件中的函数

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

建议：使用 C++版本的 C 标准库头文件

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

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

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

处理每个字符？使用基于范围的 for 语句

C++11

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

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

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

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

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

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

for 循环把变量 c 和 str 联系了起来，其中我们定义循环控制变量的方式与定义任意一个普通变量是一样的。此例中，通过使用 auto 关键字（参见 2.5.2 节，第 61 页）让编译器来决定变量 c 的类型，这里 c 的类型是 char。每次迭代，str 的下一个字符被拷贝给 c，因此该循环可以读作“对于字符串 str 中的每个字符 c，”执行某某操作。此例中的“某某操作”即输出一个字符，然后换行。 92

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

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

程序的输出结果将是：

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

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

使用范围 for 语句改变字符串中的字符 93

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

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

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

上述代码的输出结果将是：

```
HELLO WORLD!!!
```

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

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

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

只处理一部分字符？

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

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

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

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



string 对象的下标必须大于等于 0 而小于 s.size()。

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

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

下面的程序使用下标运算符输出 string 对象中的第一个字符：

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

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

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

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

程序的输出结果将是：

```
Some string
```

使用下标执行迭代

另一个例子是把 s 的第一个词改成大写形式：

```
// 依次处理 s 中的字符直至我们处理完全部字符或者遇到一个空白
for (decltype(s.size()) index = 0;
     index != s.size() && !isspace(s[index]); ++index)
         s[index] = toupper(s[index]); //将当前字符改成大写形式
```

程序的输出结果将是：

```
SOME string
```

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

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

提示：注意检查下标的合法性 95

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



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

使用下标执行随机访问

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

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

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

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

假设输入的内容如下：

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

程序的输出结果将是：

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

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

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

96

3.2.3 节练习

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

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

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

练习 3.9：下面的程序有何作用？它合法吗？如果不合法，为什么？

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

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

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

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

3.3 标准库类型 vector

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

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

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

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

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

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

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

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

97

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



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

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

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



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

3.3.1 定义和初始化 vector 对象



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

表 3.4：初始化 vector 对象的方法

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

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

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

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

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

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

98 列表初始化 vector 对象

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

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

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

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

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

创建指定数量的元素

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

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

值初始化

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

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

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

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

其二，如果只提供了元素的数量而没有设定初始值，只能使用直接初始化：

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

99

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

列表初始值还是元素数量？



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

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

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

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

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

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

100

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

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

3.3.1 节练习

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

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

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

```
(a) vector<int> v1;                    (b) vector<int> v2(10);
(c) vector<int> v3(10, 42);            (d) vector<int> v4{10};
(e) vector<int> v5{10, 42};            (f) vector<string> v6{10};
(g) vector<string> v7{10, "hi"};
```



3.3.2 向 vector 对象中添加元素

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

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

101

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

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

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

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

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

关键概念：vector 对象能高效增长

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

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

向 vector 对象添加元素蕴含的编程假定

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

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



范围 for 语句体内不应改变其所遍历序列的大小。

3.3.2 节练习

102

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

练习 3.15：改写上题的程序，不过这次读入的是字符串。

3.3.3 其他 vector 操作



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

表 3.5：vector 支持的操作

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

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

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

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

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



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

```
vector<int>::size_type        // 正确
vector::size_type             // 错误
```

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

103

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

计算 vector 内对象的索引

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

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

按照上面的描述，如果输入的成绩如下：

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

则输出的结果应该是：

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

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

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

104

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

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

执行计数值累加的那条语句很好地体现了 C++程序代码的简洁性。表达式

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

等价于

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

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

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

不能用下标形式添加元素

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

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

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

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

105

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



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

提示：只能对确知已存在的元素执行下标操作！

关于下标必须明确的一点是：只能对确知已存在的元素执行下标操作。例如，

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

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

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

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



确保下标合法的一种有效手段就是尽可能使用范围 for 语句。

3.3.3 节练习

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

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

练习 3.18：下面的程序合法吗？如果不合法，你准备如何修改？

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

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

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

3.4 迭代器介绍



我们已经知道可以使用下标运算符来访问 string 对象的字符或 vector 对象的元素，还有另外一种更通用的机制也可以实现同样的目的，这就是**迭代器**（iterator）。在第 II 部分中将要介绍，除了 vector 之外，标准库还定义了其他几种容器。所有标准库容器都可以使用迭代器，但是其中只有少数几种才同时支持下标运算符。严格来说，string 对象不属于容器类型，但是 string 支持很多与容器类型类似的操作。vector 支持下标运算符，这点和 string 一样；string 支持迭代器，这也和 vector 是一样的。 106

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

3.4.1 使用迭代器



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

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

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



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

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

迭代器运算符

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

表 3.6：标准容器迭代器的运算符	
`*iter`	返回迭代器 iter 所指元素的引用
`iter->mem`	解引用 iter 并获取该元素的名为 mem 的成员，等价于 `(*iter).mem`
`++iter`	令 iter 指示容器中的下一个元素
`--iter`	令 iter 指示容器中的上一个元素
`iter1 == iter2`	判断两个迭代器是否相等（不相等），如果两个迭代器指示的是同一个元素或者它们是同一个容器的尾后迭代器，则相等；反之，不相等
`iter1 != iter2`	

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

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

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

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

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

```
Some string
```

将迭代器从一个元素移动到另外一个元素

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



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

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

108
```
// 依次处理 s 的字符直至我们处理完全部字符或者遇到空白
for (auto it = s.begin(); it != s.end() && !isspace(*it); ++it)
    *it = toupper(*it); // 将当前字符改成大写形式
```

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

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

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

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

关键概念：泛型编程

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

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

迭代器类型

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

```
vector<int>::iterator it;     // it 能读写 vector<int>的元素
string::iterator it2;         // it2 能读写 string 对象中的字符

vector<int>::const_iterator it3; // it3 只能读元素，不能写元素
string::const_iterator it4;      // it4 只能读字符，不能写字符
```

const_iterator 和常量指针（参见 2.4.2 节，第 56 页）差不多，能读取但不能修改它所指的元素值。相反，iterator 的对象可读可写。如果 vector 对象或 string 对象是一个常量，只能使用 const_iterator；如果 vector 对象或 string 对象不是常量，那么既能使用 iterator 也能使用 const_iterator。

术语：迭代器和迭代器类型

109

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

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

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

begin 和 end 运算符

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

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

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

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

C++ 11

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

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

结合解引用和成员访问操作

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

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

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

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

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

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

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

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

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

某些对 vector 对象的操作会使迭代器失效

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

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



谨记，但凡是使用了迭代器的循环体，都不要向迭代器所属的容器添加元素。

3.4.1 节练习

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

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

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



3.4.2 迭代器运算

迭代器的递增运算令迭代器每次移动一个元素，所有的标准库容器都有支持递增运算的迭代器。类似的，也能用==和!=对任意标准库类型的两个有效迭代器（参见 3.4 节，第 95 页）进行比较。 111

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

表 3.7：vector 和 string 迭代器支持的运算

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

迭代器的算术运算

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

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

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

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

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

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

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

使用迭代器运算

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

下面的程序使用迭代器完成了二分搜索：

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

程序的一开始定义了三个迭代器：beg 指向搜索范围内的第一个元素、end 指向尾元素的下一位置、mid 指向中间的那个元素。初始状态下，搜索范围是名为 text 的 vector<string>的全部范围。

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

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

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

3.4.2 节练习

113

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

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

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

3.5 数组

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



如果不清楚元素的确切个数，请使用 vector。

3.5.1 定义和初始化内置数组

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

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

默认情况下，数组的元素被默认初始化（参见 2.2.1 节，第 40 页）。



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

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

114

显式初始化数组元素

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

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

字符数组的特殊性

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

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

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

不允许拷贝和赋值

不能将数组的内容拷贝给其他数组作为其初始值，也不能用数组为其他数组赋值：

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



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

理解复杂的数组声明

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

115

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

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

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

当然，对修饰符的数量并没有特殊限制：

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

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



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

3.5.1 节练习

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

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

练习 3.28：下列数组中元素的值是什么？

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

练习 3.29：相比于 vector 来说，数组有哪些缺点，请列举一些。

3.5.2 访问数组元素

116

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

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

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

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

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

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

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

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

检查下标的值

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



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

117

3.5.2 节练习

练习 3.30：指出下面代码中的索引错误。

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

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

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

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

3.5.3 指针和数组

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

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

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

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

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



在大多数表达式中，使用数组类型的对象其实是使用一个指向该数组首元素的指针。

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

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

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

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

118

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

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

指针也是迭代器

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

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

就像使用迭代器遍历vector对象中的元素一样，使用指针也能遍历数组中的元素。当然，这样做的前提是先得获取到指向数组第一个元素的指针和指向数组尾元素的下一位置的指针。之前已经介绍过，通过数组名字或者数组中首元素的地址都能得到指向首元素的指针；不过获取尾后指针就要用到数组的另外一个特殊性质了。我们可以设法获取数组

尾元素之后的那个并不存在的元素的地址：

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

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

利用上面得到的指针能重写之前的循环，令其输出 arr 的全部元素：

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

标准库函数 begin 和 end

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

C++11

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

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

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

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

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



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

指针运算

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

给（从）一个指针加上（减去）某整数值，结果仍是指针。新指针指向的元素与原来

的指针相比前进了（后退了）该整数值个位置：

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

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

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

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

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

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

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

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

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

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

如果两个指针分别指向不相关的对象，则不能比较它们：

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

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

解引用和指针运算的交互

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

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

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

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

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

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

下标和指针

121

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

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

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

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

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

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

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



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

3.5.3 节练习

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

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

练习3.35：编写一段程序，利用指针将数组中的元素置为0。

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

3.5.4 C 风格字符串





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

122

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

C 标准库 String 函数

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

表 3.8：C 风格字符串的函数

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



表 3.8 所列的函数不负责验证其字符串参数。

传入此类函数的指针必须指向以空字符作为结束的数组：

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

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

比较字符串

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

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

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

123

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

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

指针指向的并非同一对象，所以将得到未定义的结果。

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

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

目标字符串的大小由调用者指定

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

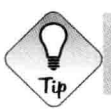
```
// 将 largeStr 初始化成 s1、一个空格和 s2 的连接
string largeStr = s1 + " " + s2;
```

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

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

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

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



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

124

3.5.4 节练习

练习 3.37：下面的程序是何含义，程序的输出结果是什么？

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

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

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

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

3.5.5 与旧代码的接口

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

混用 string 对象和 C 风格字符串



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

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

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

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

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

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

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

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



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

使用数组初始化 vector 对象

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

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

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

一样。

用于初始化 vector 对象的值也可能仅是数组的一部分：

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

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

建议：尽量使用标准库类型而非数组

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

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

3.5.5 节练习

练习 3.41：编写一段程序，用整型数组初始化一个 vector 对象。

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

3.6 多维数组

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

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

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

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

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

对于二维数组来说，常把第一个维度称作行，第二个维度称作列。

多维数组的初始化

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

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

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

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

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

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

127

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

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

含义发生了变化，它初始化的是第一行的 4 个元素，其他元素被初始化为 0。

多维数组的下标引用

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

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

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

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

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

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

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

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

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

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

使用范围 for 语句处理多维数组

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

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

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

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

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

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

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

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



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

指针和多维数组

当程序使用多维数组的名字时，也会自动将其转换成指向数组首元素的指针。



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

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

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

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



在上述声明中，圆括号必不可少：

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

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

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

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

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

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

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

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

类型别名简化多维数组的指针

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

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

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

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

3.6 节练习

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

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

练习 3.45：再一次改写程序，这次使用 auto 关键字。

小结

131

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

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

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

术语表

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

string 是一种标准库类型，表示字符的序列。

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

```
using 命名空间 :: 名字;
```

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

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

vector 是一种标准库类型，容纳某指定类型的一组元素。

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

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

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

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

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

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

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

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

第 4 章
表达式

内容

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

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

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

4.1 基础

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

4.1.1 基本概念

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

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

组合运算符和运算对象

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

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

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

运算对象转换

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

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

135 #### 重载运算符

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

符都是重载的运算符。

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

左值和右值



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

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

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

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

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

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

136

4.1.2 优先级与结合律



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

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

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

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

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

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

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

括号无视优先级与结合律

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

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

优先级与结合律有何影响

137

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

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

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

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

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

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

4.1.2 节练习

练习 4.1：表达式 5+10*20/2 的求值结果是多少？

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

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

4.1.3 求值顺序



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

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

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

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

138

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

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

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

求值顺序、优先级、结合律



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

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

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

建议：处理复合表达式

139

以下两条经验准则对书写复合表达式有益：

1. 拿不准的时候最好用括号来强制让表达式的组合关系符合程序逻辑的要求。

2. 如果改变了某个运算对象的值，在表达式的其他地方不要再使用这个运算对象。

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

4.1.3 节练习

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

4.2 算术运算符

表 4.1：算术运算符（左结合律）

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

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

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

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

一元负号运算符对运算对象值取负后，返回其（提升后的）副本：

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

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

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

提示：溢出和其他算术运算异常

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

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

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

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

```
short_value: -32768
```

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

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

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

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

141

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

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

C++11

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

```
21 % 6;      /* 结果是 3 */        21 / 6;      /* 结果是 3 */
21 % 7;      /* 结果是 0 */        21 / 7;      /* 结果是 3 */
-21 % -8;    /* 结果是-5 */        -21 / -8;    /* 结果是 2 */
21 % -5;     /* 结果是 1 */        21 / -5;     /* 结果是-4 */
```

142

4.2 节练习

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

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

练习 4.5：写出下列表达式的求值结果。

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

练习 4.6：写出一条表达式用于确定一个整数是奇数还是偶数。

练习 4.7：溢出是何含义？写出三条将导致溢出的表达式。

4.3 逻辑和关系运算符

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

表 4.2：逻辑运算符和关系运算符

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

逻辑与和逻辑或运算符

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

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

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

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

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

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

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

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

输出当前元素后检查是否需要换行。if 语句的条件部分首先检查 s 是否是一个空 string，如果是，则不论右侧运算对象的值如何都应该换行。只有当 string 对象非空时才需要求第二个运算对象的值，也就是检查 string 对象是否是以句号结束的。在这条表达式中，利用逻辑或运算符的短路求值策略确保只有当 s 非空时才会用下标运算符去访问它。 143

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

逻辑非运算符

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

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

子表达式

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

当 empty 函数返回假时结果为真。

关系运算符

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

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

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

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

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

相等性测试与布尔字面值

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

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

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

有时会试图将上面的真值测试写成如下形式：

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

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

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

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

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



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

4.3 节练习

练习 4.8：说明在逻辑与、逻辑或及相等性运算符中运算对象求值的顺序。

练习 4.9：解释在下面的 if 语句中条件部分的判断过程。

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

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

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

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

4.4 赋值运算符

赋值运算符的左侧运算对象必须是一个可修改的左值。如果给定 145

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

则下面的赋值语句都是非法的：

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

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

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

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

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

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

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

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

赋值运算满足右结合律

赋值运算符满足右结合律，这一点与其他二元运算符不太一样：

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

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

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

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

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

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

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

赋值运算优先级较低

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

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

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

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

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

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



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

切勿混淆相等运算符和赋值运算符

C++语言允许用赋值运算作为条件，但是这一特性可能带来意想不到的后果：

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

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

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

程序的这种缺陷显然很难被发现，好在一部分编译器会对类似的代码给出警告信息。

147 ### 复合赋值运算符

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

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

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

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

任意一种复合运算符都完全等价于

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

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

4.4 节练习

练习 4.13：在下述语句中，当赋值完成后 i 和 d 的值分别是多少？

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

练习 4.14：执行下述 if 语句后将发生什么情况？

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

练习 4.15：下面的赋值是非法的，为什么？应该如何修改？

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

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

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

4.5 递增和递减运算符

递增运算符（++）和递减运算符（--）为对象的加 1 和减 1 操作提供了一种简洁的书写形式。这两个运算符还可应用于迭代器，因为很多迭代器本身不支持算术运算，所以此时递增和递减运算符除了书写简洁外还是必须的。 148

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

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

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

建议：除非必须，否则不用递增递减运算符的后置版本

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

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

在一条语句中混用解引用和递增运算符

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

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

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

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

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

149

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

建议：简洁可以成为一种美德

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

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

要比书写下面的等价语句更简洁、也更少出错

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

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

运算对象可按任意顺序求值

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

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

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

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

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

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

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

也可能采取别的什么方式处理它。

4.5 节练习

150

练习 4.17：说明前置递增运算符和后置递增运算符的区别。

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

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

(a) ptr != 0 && *ptr++　　(b) ival++ && ival

(c) vec[ival++] <= vec[ival]

4.6 成员访问运算符

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

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

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

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

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

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

4.6 节练习

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

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

151 4.7 条件运算符

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

cond ? expr1 : expr2;

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

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

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

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

嵌套条件运算符

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

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

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

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



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

在输出表达式中使用条件运算符

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

152

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

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

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

因为第三条表达式等价于下面的语句，所以它是错误的：

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

4.7 节练习

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

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

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

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

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

4.8 位运算符

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

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

153

表 4.3：位运算符（左结合律）

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

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



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

移位运算符

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

```
在下面的图例中右侧为最低位并且假定 char 占 8 位、int 占 32 位
// 0233 是八进制的字面值（参见 2.1.3 节，第 35 页）
unsigned char bits = 0233;
```

1	0	0	1	1	0	1	1

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

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

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

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

```
bits >> 3 // 向右移动 3 位，最右边的 3 位丢弃掉了
```

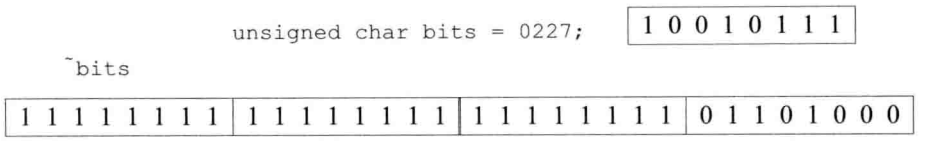
0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 1 0 0 1 1

左移运算符（<<）在右侧插入值为 0 的二进制位。**右移运算符**（>>）的行为则依赖于其左侧运算对象的类型：如果该运算对象是无符号类型，在左侧插入值为 0 的二进制位；

154 如果该运算对象是带符号类型，在左侧插入符号位的副本或值为 0 的二进制位，如何选择要视具体环境而定。

位求反运算符

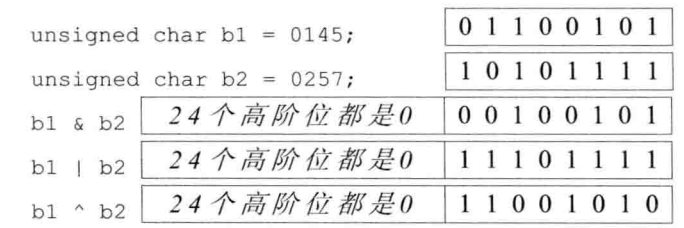
位求反运算符（~）将运算对象逐位求反后生成一个新值，将 1 置为 0、将 0 置为 1：



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

位与、位或、位异或运算符

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



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



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

使用位运算符

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

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

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

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

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

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

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

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

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

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

|=运算符的工作原理和+=非常相似，它等价于

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

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

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

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

最后，我们试图检查学生 27 测验的情况到底怎么样：

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

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



移位运算符（又叫 IO 运算符）满足左结合律

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

因为移位运算符满足左结合律，所以表达式

156

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

的执行过程实际上等同于

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

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

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

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

最后一个 cout 的含义其实是

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

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

4.8 节练习

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

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

练习 4.27：下列表达式的结果是什么？

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

4.9 sizeof 运算符

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

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

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

157

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

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

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

C++ 11

sizeof 运算符的结果部分地依赖于其作用的类型：

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

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

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

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

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

4.9 节练习

练习 4.28：编写一段程序，输出每一种内置类型所占空间的大小。

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

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

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

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

4.10 逗号运算符

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

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

逗号运算符经常被用在 for 循环当中：

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

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

4.10 节练习

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

练习 4.32：解释下面这个循环的含义。

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

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

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

4.11 类型转换

159



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

举个例子，考虑下面这条表达式，它的目的是将 `ival` 初始化为 6：

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

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

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

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

何时发生隐式类型转换

在下面这些情况下，编译器会自动地转换运算对象的类型：

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

4.11.1 算术转换

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

160 **整型提升**

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

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

无符号类型的运算对象

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

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

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

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

理解算术转换

要想理解算术转换，办法之一就是研究大量的例子：

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

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

161

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

4.11.1 节练习

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

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

需要注意每种运算符遵循的是左结合律还是右结合律。

练习 4.35：假设有如下的定义，

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

请回答在下面的表达式中发生了隐式类型转换吗？如果有，指出来。

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

4.11.2 其他隐式类型转换



除了算术转换之外还有几种隐式类型转换，包括如下几种。

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

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

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

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

162

型间还有另外一种指针转换的方式。

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

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

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

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

相反的转换并不存在，因为它试图删除掉底层const。

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

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

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

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

4.11.3 显式转换

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

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

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



虽然有时不得不使用强制类型转换，但这种方法本质上是非常危险的。

163 **命名的强制类型转换**

一个命名的强制类型转换具有如下形式：

cast-name<type>(expression);

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

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

static_cast

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

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

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

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

```
void* p = &d;    // 正确：任何非常量对象的地址都能存入 void*
//正确：将 void*转换回初始的指针类型
double *dp = static_cast<double*>(p);
```

当我们把指针存放在 void*中，并且使用 static_cast 将其强制转换回原来的类型时，应该确保指针的值保持不变。也就是说，强制转换的结果将与原始的地址值相等，因此我们必须确保转换后所得的类型就是指针所指的类型。类型一旦不符，将产生未定义的后果。

const_cast

const_cast 只能改变运算对象的底层 const（参见 2.4.3 节，第 57 页）：

```
const char *pc;
char *p = const_cast<char*>(pc); // 正确：但是通过 p 写值是未定义的行为
```

对于将常量对象转换成非常量对象的行为，我们一般称其为“去掉 const 性质（cast away the const）”。一旦我们去掉了某个对象的 const 性质，编译器就不再阻止我们对该对象进行写操作了。如果对象本身不是一个常量，使用强制类型转换获得写权限是合法的行为。然而如果对象是一个常量，再使用 const_cast 执行写操作就会产生未定义的后果。

只有 const_cast 能改变表达式的常量属性，使用其他形式的命名强制类型转换改变表达式的常量属性都将引发编译器错误。同样的，也不能用 const_cast 改变表达式的类型： 164

```
const char *cp;
// 错误：static_cast 不能转换掉 const 性质
char *q = static_cast<char*>(cp);
static_cast<string>(cp);      // 正确：字符串字面值转换成 string 类型
const_cast<string>(cp);       // 错误：const_cast 只改变常量属性
```

const_cast 常常用于有函数重载的上下文中，关于函数重载将在 6.4 节（第 208 页）进行详细介绍。

reinterpret_cast

reinterpret_cast 通常为运算对象的位模式提供较低层次上的重新解释。举个例

子，假设有如下的转换

```
int *ip;
char *pc = reinterpret_cast<char*>(ip);
```

我们必须牢记 pc 所指的真实对象是一个 int 而非字符，如果把 pc 当成普通的字符指针使用就可能在运行时发生错误。例如：

```
string str(pc);
```

可能导致异常的运行时行为。

使用 reinterpret_cast 是非常危险的，用 pc 初始化 str 的例子很好地证明了这一点。其中的关键问题是类型改变了，但编译器没有给出任何警告或者错误的提示信息。当我们用一个 int 的地址初始化 pc 时，由于显式地声称这种转换合法，所以编译器不会发出任何警告或错误信息。接下来再使用 pc 时就会认定它的值是 char*类型，编译器没法知道它实际存放的是指向 int 的指针。最终的结果就是，在上面的例子中虽然用 pc 初始化 str 没什么实际意义，甚至还可能引发更糟糕的后果，但仅从语法上而言这种操作无可指摘。查找这类问题的原因非常困难，如果将 ip 强制转换成 pc 的语句和用 pc 初始化 string 对象的语句分属不同文件就更是如此。



reinterpret_cast 本质上依赖于机器。要想安全地使用 reinterpret_cast 必须对涉及的类型和编译器实现转换的过程都非常了解。

165

建议：避免强制类型转换

强制类型转换干扰了正常的类型检查（参见 2.2.2 节，第 42 页），因此我们强烈建议程序员避免使用强制类型转换。这个建议对于 reinterpret_cast 尤其适用，因为此类类型转换总是充满了风险。在有重载函数的上下文中使用 const_cast 无可厚非，关于这一点将在 6.4 节（第 208 页）中详细介绍；但是在其他情况下使用 const_cast 也就意味着程序存在某种设计缺陷。其他强制类型转换，比如 static_cast 和 dynamic_cast，都不应该频繁使用。每次书写了一条强制类型转换语句，都应该反复斟酌能否以其他方式实现相同的目标。就算实在无法避免，也应该尽量限制类型转换值的作用域，并且记录对相关类型的所有假定，这样可以减少错误发生的机会。

旧式的强制类型转换

在早期版本的 C++语言中，显式地进行强制类型转换包含两种形式：

```
type (expr);          // 函数形式的强制类型转换
(type) expr;          // C 语言风格的强制类型转换
```

根据所涉及的类型不同，旧式的强制类型转换分别具有与 const_cast、static_cast 或 reinterpret_cast 相似的行为。当我们在某处执行旧式的强制类型转换时，如果换成 const_cast 和 static_cast 也合法，则其行为与对应的命名转换一致。如果替换后不合法，则旧式强制类型转换执行与 reinterpret_cast 类似的功能：

```
char *pc = (char*) ip; // ip 是指向整数的指针
```

的效果与使用 reinterpret_cast 一样。



与命名的强制类型转换相比，旧式的强制类型转换从表现形式上来说不那么清晰明了，容易被看漏，所以一旦转换过程出现问题，追踪起来也更加困难。

4.11.3 节练习

练习 4.36：假设 i 是 int 类型，d 是 double 类型，书写表达式 i*=d 使其执行整数类型的乘法而非浮点类型的乘法。

练习 4.37：用命名的强制类型转换改写下列旧式的转换语句。

```
int i; double d; const string *ps; char *pc; void *pv;
(a) pv = (void*)ps;    (b) i = int(*pc);
(c) pv = &d;           (d) pc = (char*) pv;
```

练习 4.38：说明下面这条表达式的含义。

```
double slope = static_cast<double>(j/i);
```

4.12 运算符优先级表

166

表 4.4：运算符优先级

结合律	和运算符	功能	用法	参考页码
左	::	全局作用域	::name	256
左	::	类作用域	class::name	79
左	::	命名空间作用域	namespace::name	74
左	.	成员选择	object.member	20
左	->	成员选择	pointer->member	98
左	[]	下标	expr[expr]	104
左	()	函数调用	name(expr_list)	20
左	()	类型构造	type(expr_list)	145
右	++	后置递增运算	lvalue++	131
右	--	后置递减运算	lvalue--	131
右	typeid	类型 ID	typeid(type)	731
右	typeid	运行时类型 ID	typeid(expr)	731
右	explicit cast	类型转换	*cast_name*<type>(expr)	144
右	++	前置递增运算	++lvalue	131
右	--	前置递减运算	--lvalue	131
右	~	位求反	~expr	136
右	!	逻辑非	!expr	126
右	-	一元负号	-expr	124
右	+	一元正号	+expr	124
右	*	解引用	*expr	48
右	&	取地址	&lvalue	47
右	()	类型转换	(type) expr	145
右	sizeof	对象的大小	sizeof expr	139

续表

结合律	和运算符	功能	用法	参考页码
右	sizeof	类型的大小	sizeof(type)	139
右	Sizeof...	参数包的大小	sizeof...(name)	619
右	new	创建对象	new type	407
右	new[]	创建数组	new type[size]	407
右	delete	释放对象	delete expr	409
右	delete[]	释放数组	delete[] expr	409
右	noexcept	能否抛出异常	noexcept (expr)	690
左	->*	指向成员选择的指针	ptr->*ptr_to_member	740
左	.*	指向成员选择的指针	obj.*ptr_to_member	740
左	*	乘法	expr * expr	124
左	/	除法	expr / expr	124
左	%	取模（取余）	expr % expr	124
左	+	加法	expr + expr	124
左	-	减法	expr - expr	124
左	<<	向左移位	expr << expr	136
左	>>	向右移位	expr >> expr	136
左	<	小于	expr < expr	126
左	<=	小于等于	expr <= expr	126
左	>	大于	expr > expr	126
左	>=	大于等于	expr >= expr	126
左	==	相等	expr == expr	126
左	!=	不相等	expr != expr	126
左	&	位与	expr & expr	136
左	^	位异或	expr ^ expr	136
左	|	位或	expr | expr	136
左	&&	逻辑与	expr && expr	126
左	||	逻辑或	expr || expr	126
右	? :	条件	expr ? expr : expr	134
右	=	赋值	lvalue = expr	129
右	*=, /=, %=	复合赋值	lvalue += expr 等	129
右	+=, -=			129
右	<<=, >>=			129
右	&=, |=, ^=			129
右	throw	抛出异常	throw expr	173
左	,	逗号	expr, expr	140

167

小结

168

C++语言提供了一套丰富的运算符，并定义了这些运算符作用于内置类型的运算对象时所执行的操作。此外，C++语言还支持运算符重载的机制，允许我们自己定义运算符作用于类类型时的含义。第 14 章将介绍如何定义作用于用户类型的运算符。

对于含有超过一个运算符的表达式，要想理解其含义关键要理解优先级、结合律和求值顺序。每个运算符都有其对应的优先级和结合律，优先级规定了复合表达式中运算符组合的方式，结合律则说明当运算符的优先级一样时应该如何组合。

大多数运算符并不明确规定运算对象的求值顺序：编译器有权自由选择先对左侧运算对象求值还是先对右侧运算对象求值。一般来说，运算对象的求值顺序对表达式的最终结果没有影响。但是，如果两个运算对象指向同一个对象而且其中一个改变了对象的值，就会导致程序出现不易发现的严重缺陷。

最后一点，运算对象经常从原始类型自动转换成某种关联的类型。例如，表达式中的小整型会自动提升成大整型。不论内置类型还是类类型都涉及类型转换的问题。如果需要，我们还可以显式地进行强制类型转换。

术语表

算术转换（arithmetic conversion）从一种算术类型转换成另一种算术类型。在二元运算符的上下文中，为了保留精度，算术转换通常把较小的类型转换成较大的类型（例如整型转换成浮点型）。

结合律（associativity）规定具有相同优先级的运算符如何组合在一起。结合律分为左结合律（运算符从左向右组合）和右结合律（运算符从右向左组合）。

二元运算符（binary operator）有两个运算对象参与运算的运算符。

强制类型转换（cast）一种显式的类型转换。

复合表达式（compound expression）含有多于一个运算符的表达式。

const_cast 一种涉及 `const` 的强制类型转换。将底层 `const` 对象转换成对应的非常量类型，或者执行相反的转换。

转换（conversion）一种类型的值改变成另一种类型的值的过程。C++语言定义了内置类型的转换规则。类类型同样可以转换。

dynamic_cast 和继承及运行时类型识别一起使用。参见 19.2 节（第 730 页）。

表达式（expression）C++程序中最低级别的计算。表达式将运算符作用于一个或多个运算对象，每个表达式都有对应的求值结果。表达式本身也可以作为运算对象，这时就得到了对多个运算符求值的复合表达式。

隐式转换（implicit conversion）由编译器自动执行的类型转换。假如表达式需要某种特定的类型而运算对象是另外一种类型，此时只要规则允许，编译器就会自动地将运算对象转换成所需的类型。

整型提升（integral promotion） 把一种较小的整数类型转换成与之最接近的较大整数类型的过程。不论是否真的需要，小整数类型（即 `short`、`char` 等）总是会得到提升。

169

左值（lvalue）是指那些求值结果为对象或函数的表达式。一个表示对象的非常量左值可以作为赋值运算符的左侧运算对象。

运算对象（operand）表达式在某些值上执行运算，这些值就是运算对象。一个运算

符有一个或多个相关的运算对象。

运算符（operator）决定表达式所做操作的符号。C++语言定义了一套运算符并说明了这些运算符作用于内置类型时的含义。C++还定义了运算符的优先级和结合律以及每种运算符处理的运算对象数量。可以重载运算符使其能处理类类型。

求值顺序（order of evaluation）是某个运算符的运算对象的求值顺序。大多数情况下，编译器可以任意选择运算对象求值的顺序。不过运算对象一定要在运算符之前得到求值结果。只有&&、||、条件和逗号四种运算符明确规定了求值顺序。

重载运算符（overloaded operator）针对某种运算符重新定义的适用于类类型的版本。第14章将介绍重载运算符的方法。

优先级（precedence）规定了复合表达式中不同运算符的执行顺序。与低优先级的运算符相比，高优先级的运算符组合得更紧密。

提升（promoted）参见整型提升。

reinterpret_cast 把运算对象的内容解释成另外一种类型。这种强制类型转换本质上依赖于机器而且非常危险。

结果（result）计算表达式得到的值或对象。

右值（rvalue）是指一种表达式，其结果是值而非值所在的位置。

短路求值（short-circuit evaluation） 是一个专有名词，描述逻辑与运算符和逻辑或运算符的执行过程。如果根据运算符的第一个运算对象就能确定整个表达式的结果，求值终止，此时第二个运算对象将不会被求值。

sizeof 是一个运算符，返回存储对象所需的字节数，该对象的类型可能是某个给定的类型名字，也可能由表达式的返回结果确定。

static_cast 显式地执行某种定义明确的类型转换，常用于替代由编译器隐式执行的类型转换。

一元运算符（unary operators）只有一个运算对象参与运算的运算符。

, 运算符（, operator）逗号运算符，是一种从左向右求值的二元运算符。逗号运算符的结果是右侧运算对象的值，当且仅当右侧运算对象是左值时逗号运算符的结果是左值。

? :运算符（?: operator）条件运算符，以下述形式提供 if-then-else 逻辑的表达式

```
cond ? expr1 : expr2;
```

如果条件 *cond* 为真，对 *expr1* 求值；否则对 *expr2* 求值。*expr1* 和 *expr2* 的类型应该相同或者能转换成同一种类型。*expr1* 和 *expr2* 中只有一个会被求值。

&&运算符（&& operator）逻辑与运算符，如果两个运算对象都是真，结果才为真。只有当左侧运算对象为真时才会检查右侧运算对象。

&运算符（& operator）位与运算符，由两个运算对象生成一个新的整型值。如果两个运算对象对应的位都是1，所得结果中该位为1；否则所得结果中该位为0。

170

^运算符（^ operator）位异或运算符，由两个运算对象生成一个新的整型值。如果两个运算对象对应的位有且只有一个是1，所得结果中该位为1；否则所得结果中该位为0。

||运算符（|| operator）逻辑或运算符，任何一个运算对象是真，结果就为真。只有当左侧运算对象为假时才会检查右侧运算对象。

| 运算符（| operator）位或运算符，由两个运算对象生成一个新的整型值。如果两个运算对象对应的位至少有一个是1，所得结果中该位为1；否则所得结果中该位为0。

++运算符（++ operator）递增运算符。包括两种形式：前置版本和后置版本。前置递增运算符得到一个左值，它给运算符加1并得到运算对象改变后的值。后置递增运算符得到一个右值，它给运算符加1并得到运算对象原始的、未改变的值的副本。注意：即使迭代器没有定义+运算符，也会

有++运算符。

—运算符（— operator）递减运算符。包括两种形式：前置版本和后置版本。前置递减运算符得到一个左值，它从运算符减 1 并得到运算对象改变后的值。后置递减运算符得到一个右值，它从运算符减 1 并得到运算对象原始的、未改变的值的副本。注意：即使迭代器没有定义–运算符，也会有––运算符。

<<运算符（<< operator）左移运算符，将左侧运算对象的值的（可能是提升后的）副本向左移位，移动的位数由右侧运算对象确定。右侧运算对象必须大于等于 0 而且小于结果的位数。左侧运算对象应该是无符号类型，如果它是带符号类型，则一旦移动改变了符号位的值就会产生未定义的结果。

>>运算符（>> operator）右移运算符，除了移动方向相反，其他性质都和左移运算符类似。如果左侧运算对象是带符号类型，那么根据实现的不同新移入的内容也不同，新移入的位可能都是 0，也可能都是符号位的副本。

~运算符（~ operator）位求反运算符，生成一个新的整型值。该值的每一位恰好与（可能是提升后的）运算对象的对应位相反。

!运算符（! operator）逻辑非运算符，将它的运算对象的布尔值取反。如果运算对象是假，则结果为真，如果运算对象是真，则结果为假。

第5章
语句

内容

5.1 简单语句 154
5.2 语句作用域 155
5.3 条件语句 156
5.4 迭代语句 165
5.5 跳转语句 170
5.6 try 语句块和异常处理 172
小结 178
术语表 178

和大多数语言一样，C++提供了条件执行语句、重复执行相同代码的循环语句和用于中断当前控制流的跳转语句。本章将详细介绍 C++语言所支持的这些语句。

172 通常情况下，语句是顺序执行的。但除非是最简单的程序，否则仅有顺序执行远远不够。因此，C++语言提供了一组控制流（flow-of-control）语句以支持更复杂的执行路径。

5.1 简单语句

C++语言中的大多数语句都以分号结束，一个表达式，比如 `ival + 5`，末尾加上分号就变成了**表达式语句**（expression statement）。表达式语句的作用是执行表达式并丢弃掉求值结果：

```
ival + 5;          // 一条没什么实际用处的表达式语句
cout << ival;      // 一条有用的表达式语句
```

第一条语句没什么用处，因为虽然执行了加法，但是相加的结果没被使用。比较普遍的情况是，表达式语句中的表达式在求值时附带有其他效果，比如给变量赋了新值或者输出了结果。

空语句

最简单的语句是**空语句**（null statement），空语句中只含有一个单独的分号：

```
; // 空语句
```

如果在程序的某个地方，语法上需要一条语句但是逻辑上不需要，此时应该使用空语句。一种常见的情况是，当循环的全部工作在条件部分就可以完成时，我们通常会用到空语句。例如，我们想读取输入流的内容直到遇到一个特定的值为止，除此之外什么事情也不做：

```
// 重复读入数据直至到达文件末尾或某次输入的值等于 sought
while (cin >> s && s != sought)
    ; // 空语句
```

`while` 循环的条件部分首先从标准输入读取一个值并且隐式地检查 `cin`，判断读取是否成功。假定读取成功，条件的后半部分检查读进来的值是否等于 `sought` 的值。如果发现了想要的值，循环终止；否则，从 `cin` 中继续读取另一个值，再一次判断循环的条件。



使用空语句时应该加上注释，从而令读这段代码的人知道该语句是有意省略的。

别漏写分号，也别多写分号

因为空语句是一条语句，所以可用在任何允许使用语句的地方。由于这个原因，某些看起来非法的分号往往只不过是一条空语句而已，从语法上说得过去。下面的片段包含两条语句：表达式语句和空语句。

173
```
ival = v1 + v2;; // 正确：第二个分号表示一条多余的空语句
```

多余的空语句一般来说是无害的，但是如果在 `if` 或者 `while` 的条件后面跟了一个额外的分号就可能完全改变程序员的初衷。例如，下面的代码将无休止地循环下去：

```
// 出现了糟糕的情况：额外的分号，循环体是那条空语句
while (iter != svec.end()) ;      // while 循环体是那条空语句
    ++iter;                       // 递增运算不属于循环的一部分
```

虽然从形式上来看执行递增运算的语句前面有缩进，但它并不是循环的一部分。循环条件后面跟着的分号构成了一条空语句，它才是真正的循环体。



多余的空语句并非总是无害的。

复合语句（块）

复合语句（compound statement）是指用花括号括起来的（可能为空的）语句和声明的序列，复合语句也被称作**块**（block）。一个块就是一个作用域（参见 2.2.4 节，第 43 页），在块中引入的名字只能在块内部以及嵌套在块中的子块里访问。通常，名字在有限的区域内可见，该区域从名字定义处开始，到名字所在的（最内层）块的结尾为止。

如果在程序的某个地方，语法上需要一条语句，但是逻辑上需要多条语句，则应该使用复合语句。例如，while 或者 for 的循环体必须是一条语句，但是我们常常需要在循环体内做很多事情，此时就需要将多条语句用花括号括起来，从而把语句序列转变成块。

举个例子，回忆 1.4.1 节（第 10 页）的 while 循环：

```
while (val <= 10) {
    sum += val;       // 把 sum + val 的值赋给 sum。
    ++val;            // 给 val 加 1
}
```

程序从逻辑上来说要执行两条语句，但是 while 循环只能容纳一条。此时，把要执行的语句用花括号括起来，就将其转换成了一条（复合）语句。



块不以分号作为结束。

所谓空块，是指内部没有任何语句的一对花括号。空块的作用等价于空语句：

```
while (cin >> s && s != sought)
    { } // 空块
```

5.1 节练习

174

练习 5.1：什么是空语句？什么时候会用到空语句？

练习 5.2：什么是块？什么时候会用到块？

练习 5.3：使用逗号运算符（参见 4.10 节，第 140 页）重写 1.4.1 节（第 10 页）的 while 循环，使它不再需要块，观察改写之后的代码的可读性提高了还是降低了。

5.2 语句作用域

可以在 if、switch、while 和 for 语句的控制结构内定义变量。定义在控制结构当中的变量只在相应语句的内部可见，一旦语句结束，变量也就超出其作用范围了：

```
while (int i = get_num()) // 每次迭代时创建并初始化 i
    cout << i << endl;
i = 0; // 错误：在循环外部无法访问 i
```

如果其他代码也需要访问控制变量，则变量必须定义在语句的外部：

```
// 寻找第一个负值元素
auto beg = v.begin();
while (beg != v.end() && *beg >= 0)
    ++beg;
if (beg == v.end())
    // 此时我们知道 v 中的所有元素都大于等于 0
```

因为控制结构定义的对象的值马上要由结构本身使用，所以这些变量必须初始化。

5.2 节练习

练习 5.4：说明下列例子的含义，如果存在问题，试着修改它。

```
(a) while (string::iterator iter != s.end()) { /* ...*/ }
(b) while (bool status = find(word)) { /* ...*/ }
    if (!status) { /* ...*/ }
```

5.3 条件语句

C++语言提供了两种按条件执行的语句。一种是 `if` 语句，它根据条件决定控制流；另外一种是 `switch` 语句，它计算一个整型表达式的值，然后根据这个值从几条执行路径中选择一条。



5.3.1 if 语句

175

if 语句（if statement）的作用是：判断一个指定的条件是否为真，根据判断结果决定是否执行另外一条语句。if 语句包括两种形式：一种含有 else 分支，另外一种没有。简单 if 语句的语法形式是

```
if (condition)
    statement
```

if else 语句的形式是

```
if (condition)
    statement
else
    statement2
```

在这两个版本的 `if` 语句中，*condition* 都必须用圆括号包围起来。*condition* 可以是一个表达式，也可以是一个初始化了的变量声明（参见 5.2 节，第 155 页）。不管是表达式还是变量，其类型都必须能转换成（参见 4.11 节，第 141 页）布尔类型。通常情况下，*statement* 和 *statement2* 是块语句。

如果 *condition* 为真，执行 *statement*。当 *statement* 执行完成后，程序继续执行 `if` 语句后面的其他语句。

如果 *condition* 为假，跳过 *statement*。对于简单 `if` 语句来说，程序继续执行 `if` 语句后面的其他语句；对于 `if else` 语句来说，执行 *statement2*。

使用 if else 语句

我们举个例子来说明 if 语句的功能，程序的目的是把数字形式表示的成绩转换成字母形式。假设数字成绩的范围是从 0 到 100（包括 100 在内），其中 100 分对应的字母形式是“A++”，低于 60 分的成绩对应的字母形式是“F”。其他成绩每 10 个划分成一组：60 到 69（包括 69 在内）对应字母“D”、70 到 79 对应字母“C”，以此类推。使用 vector 对象存放字母成绩所有可能的取值：

```
const vector<string> scores = {"F", "D", "C", "B", "A", "A++"};
```

我们使用 if else 语句解决该问题，根据成绩是否合格执行不同的操作：

```
// 如果 grade 小于 60，对应的字母是 F；否则计算其下标
string lettergrade;
if (grade < 60)
    lettergrade = scores[0];
else
    lettergrade = scores[(grade - 50)/10];
```

判断 grade 的值是否小于 60，根据结果选择执行 if 分支还是 else 分支。在 else 分支中，由成绩计算得到一个下标，具体过程是：首先从 grade 中减去 50，然后执行整数除法（参见 4.2 节，在 125 页），去掉余数后所得的商就是数组 scores 对应的下标。

嵌套 if 语句

176

接下来让我们的程序更有趣点儿，试着给那些合格的成绩后面添加一个加号或减号。如果成绩的末位是 8 或者 9，添加一个加号；如果末位是 0、1 或 2，添加一个减号：

```
if (grade % 10 > 7)
    lettergrade += '+'; // 末尾是 8 或者 9 的成绩添加一个加号
else if (grade % 10 < 3)
    lettergrade += '-'; // 末尾是 0、1 或者 2 的成绩添加一个减号
```

我们使用取模运算符（参见 4.2 节，第 125 页）计算余数，根据余数决定添加哪种符号。

接着把这段添加符号的代码整合到转换成绩形式的代码中去：

```
// 如果成绩不合格，不需要考虑添加加号减号的问题
if (grade < 60)
    lettergrade = scores[0];
else {
    lettergrade = scores[(grade - 50)/10]; // 获得字母形式的成绩
    if (grade != 100) // 只要不是 A++，就考虑添加加号或减号
        if (grade % 10 > 7)
            lettergrade += '+'; // 末尾是 8 或者 9 的成绩添加一个加号
        else if (grade % 10 < 3)
            lettergrade += '-'; // 末尾是 0、1 或者 2 的成绩添加一个减号
}
```

注意，我们使用花括号把第一个 else 后面的两条语句组合成了一个块。如果 grade 不小于 60 要做两件事：从数组 scores 中获取对应的字母成绩，然后根据条件设置加号或减号。

注意使用花括号

有一种常见的错误：本来程序中有几条语句应该作为一个块来执行，但是我们忘了用花括号把这些语句包围。在下面的例子中，添加加号减号的代码将被无条件地执行，这显然违背了我们的初衷：

```
if (grade < 60)
    lettergrade = scores[0];
else // 错误：缺少花括号
    lettergrade = scores[(grade - 50)/10];
    // 虽然下面的语句从形式上看有缩进，但是因为没有花括号，
    // 所以无论什么情况都会执行接下来的代码
    // 不及格的成绩也会添加上加号或减号，这显然是错误的
    if (grade != 100)
        if (grade % 10 > 7)
            lettergrade += '+'; // 末尾是 8 或者 9 的成绩添加一个加号
        else if (grade % 10 < 3)
            lettergrade += '-'; // 末尾是 0、1 或者 2 的成绩添加一个减号
```

要想发现这个错误可能非常困难，毕竟这段代码“看起来”是正确的。

为了避免此类问题，有些编码风格要求在 if 或 else 之后必须写上花括号（对 while 和 for 语句的循环体两端也有同样的要求）。这么做的好处是可以避免代码混乱不清，以后修改代码时如果想添加别的语句，也可以很容易地找到正确位置。

177



许多编辑器和开发环境都提供一种辅助工具，它可以自动地缩进代码以匹配其语法结构。善用此类工具益处多多。

悬垂 else

当一个 if 语句嵌套在另一个 if 语句内部时，很可能 if 分支会多于 else 分支。事实上，之前那个成绩转换的程序就有 4 个 if 分支，而只有 2 个 else 分支。这时候问题出现了：我们怎么知道某个给定的 else 是和哪个 if 匹配呢？

这个问题通常称作**悬垂 else**（dangling else），在那些既有 if 语句又有 if else 语句的编程语言中是个普遍存在的问题。不同语言解决该问题的思路也不同，就 C++而言，它规定 else 与离它最近的尚未匹配的 if 匹配，从而消除了程序的二义性。

当代码中 if 分支多于 else 分支时，程序员有时会感觉比较麻烦。举个例子来说明，对于添加加号减号的那个最内层的 if else 语句，我们用另外一组条件改写它：

```
// 错误：实际的执行过程并非像缩进格式显示的那样；else 分支匹配的是内层 if 语句
if (grade % 10 >= 3)
    if (grade % 10 > 7)
        lettergrade += '+'; //末尾是 8 或者 9 的成绩添加一个加号
else
    lettergrade += '-'; //末尾是 3、4、5、6 或者 7 的成绩添加一个减号！
```

从代码的缩进格式来看，程序的初衷应该是希望 else 和外层的 if 匹配，也就是说，我们希望当 grade 的末位小于 3 时执行 else 分支。然而，不管我们是什么意图，也不管程序如何缩进，这里的 else 分支其实是内层 if 语句的一部分。最终，上面的代码将在末位大于 3 小于等于 7 的成绩后面添加减号！它的执行过程实际上等价于如下形式：

```
// 缩进格式与执行过程相符，但不是程序员的意图
if (grade % 10 >= 3)
    if (grade % 10 > 7)
        lettergrade += '+'; // 末尾是 8 或者 9 的成绩添加一个加号
    else
        lettergrade += '-'; // 末尾是 3、4、5、6 或者 7 的成绩添加一个减号!
```

使用花括号控制执行路径

要想使 else 分支和外层的 if 语句匹配起来，可以在内层 if 语句的两端加上花括号，使其成为一个块：

```
//末尾是 8 或者 9 的成绩添加一个加号，末尾是 0、1 或者 2 的成绩添加一个减号
if (grade % 10 >= 3) {
    if (grade % 10 > 7)
        lettergrade += '+'; // 末尾是 8 或者 9 的成绩添加一个加号
} else // 花括号强迫 else 与外层 if 匹配
    lettergrade += '-';       // 末尾是 0、1 或者 2 的成绩添加一个减号
```

语句属于块，意味着语句一定在块的边界之内，因此内层 if 语句在关键字 else 前面的那个花括号处已经结束了。else 不会再作为内层 if 的一部分。此时，最近的尚未匹配的 if 是外层 if，也就是我们希望 else 匹配的那个。 178

5.3.1 节练习

练习 5.5：写一段自己的程序，使用 if else 语句实现把数字成绩转换成字母成绩的要求。

练习 5.6：改写上一题的程序，使用条件运算符（参见 4.7 节，第 134 页）代替 if else 语句。

练习 5.7：改正下列代码段中的错误。

```
(a) if (ival1 != ival2)
        ival1 = ival2
    else ival1 = ival2 = 0;

(b) if (ival < minval)
        minval = ival;
        occurs = 1;
(c) if (int ival = get_value())
        cout << "ival = " << ival << endl;
    if (!ival)
        cout << "ival = 0\n";
(d) if (ival = 0)
        ival = get_value();
```

练习 5.8：什么是“悬垂 else”？C++语言是如何处理 else 子句的？

5.3.2 switch 语句

switch 语句（switch statement）提供了一条便利的途径使得我们能够在若干固定选项中做出选择。举个例子，假如我们想统计五个元音字母在文本中出现的次数，程序逻辑应该如下所示：

- 从输入的内容中读取所有字符。
- 令每一个字符都与元音字母的集合比较。
- 如果字符与某个元音字母匹配，将该字母的数量加 1。
- 显示结果。

例如，以（原书中）本章的文本作为输入内容，程序的输出结果将是：

```
Number of vowel a: 3195
Number of vowel e: 6230
Number of vowel i: 3102
Number of vowel o: 3289
Number of vowel u: 1033
```

179 要想实现这项功能，直接使用 switch 语句即可：

```
// 为每个元音字母初始化其计数值
unsigned aCnt = 0, eCnt = 0, iCnt = 0, oCnt = 0, uCnt = 0;
char ch;
while (cin >> ch) {
    // 如果 ch 是元音字母，将其对应的计数值加 1
    switch (ch) {
        case 'a':
            ++aCnt;
            break;
        case 'e':
            ++eCnt;
            break;
        case 'i':
            ++iCnt;
            break;
        case 'o':
            ++oCnt;
            break;
        case 'u':
            ++uCnt;
            break;
    }
}
// 输出结果
cout << "Number of vowel a: \t" << aCnt << '\n'
     << "Number of vowel e: \t" << eCnt << '\n'
     << "Number of vowel i: \t" << iCnt << '\n'
     << "Number of vowel o: \t" << oCnt << '\n'
     << "Number of vowel u: \t" << uCnt << endl;
```

switch 语句首先对括号里的表达式求值，该表达式紧跟在关键字 switch 的后面，可以是一个初始化的变量声明（参见 5.2 节，第 155 页）。表达式的值转换成整数类型，然后与每个 case 标签的值比较。

如果表达式和某个 case 标签的值匹配成功，程序从该标签之后的第一条语句开始执行，直到到达了 switch 的结尾或者是遇到一条 break 语句为止。

我们将在 5.5.1 节（第 170 页）详细介绍 break 语句，简言之，break 语句的作用是

中断当前的控制流。此例中，break 语句将控制权转移到 switch 语句外面。因为 switch 是 while 循环体内唯一的语句，所以从 switch 语句中断出来以后，程序的控制权将移到 while 语句的右花括号处。此时 while 语句内部没有其他语句要执行，所以 while 会返回去再一次判断条件是否满足。

如果 switch 语句的表达式和所有 case 都没有匹配上，将直接跳转到 switch 结构之后的第一条语句。刚刚说过，在上面的例子中，退出 switch 后控制权回到 while 语句的条件部分。

case 关键字和它对应的值一起被称为 **case 标签**（case label）。case 标签必须是整型常量表达式（参见 2.4.4 节，第 58 页）：

180

```
char ch = getVal();
int ival = 42;
switch(ch) {
case 3.14: // 错误：case 标签不是一个整数
case ival: // 错误：case 标签不是一个常量
//...
```

任何两个 case 标签的值不能相同，否则就会引发错误。另外，default 也是一种特殊的 case 标签，关于它的知识将在第 162 页介绍。

switch 内部的控制流

理解程序在 case 标签之间的执行流程非常重要。如果某个 case 标签匹配成功，将从该标签开始往后顺序执行所有 case 分支，除非程序显式地中断了这一过程，否则直到 switch 的结尾处才会停下来。要想避免执行后续 case 分支的代码，我们必须显式地告诉编译器终止执行过程。大多数情况下，在下一个 case 标签之前应该有一条 break 语句。

然而，也有一些时候默认的 switch 行为才是程序真正需要的。每个 case 标签只能对应一个值，但是有时候我们希望两个或更多个值共享同一组操作。此时，我们就故意省略掉 break 语句，使得程序能够连续执行若干个 case 标签。

例如，也许我们想统计的是所有元音字母出现的总次数：

```
unsigned vowelCnt = 0;
// ...
switch (ch)
{
    // 出现了 a、e、i、o 或 u 中的任意一个都会将 vowelCnt 的值加 1
    case 'a':
    case 'e':
    case 'i':
    case 'o':
    case 'u':
        ++vowelCnt;
        break;
}
```

在上面的代码中，几个 case 标签连写在一起，中间没有 break 语句。因此只要 ch 是元音字母，不管到底是五个中的哪一个都执行相同的代码。

C++程序的形式比较自由，所以 case 标签之后不一定非得换行。把几个 case 标签

写在一行里，强调这些 case 代表的是某个范围内的值：

```
switch (ch)
{
    // 另一种合法的书写形式
    case 'a': case 'e': case 'i': case 'o': case 'u':
        ++vowelCnt;
        break;
}
```

181



一般不要省略 case 分支最后的 break 语句。如果没写 break 语句，最好加一段注释说清楚程序的逻辑。

漏写 break 容易引发缺陷

有一种常见的错觉是程序只执行匹配成功的那个 case 分支的语句。例如，下面程序的统计结果是错误的：

```
// 警告：不正确的程序逻辑！
switch (ch) {
    case 'a':
        ++aCnt; // 此处应该有一条 break 语句
    case 'e':
        ++eCnt; // 此处应该有一条 break 语句
    case 'i':
        ++iCnt; // 此处应该有一条 break 语句
    case 'o':
        ++oCnt; // 此处应该有一条 break 语句
    case 'u':
        ++uCnt;
}
```

要想理解这段程序的执行过程，不妨假设 ch 的值是'e'。此时，程序直接执行 case 'e' 标签后面的代码，该代码把 eCnt 的值加 1。接下来，程序将跨越 case 标签的边界，接着递增 iCnt、oCnt 和 uCnt。



尽管 switch 语句不是非得在最后一个标签后面写上 break，但是为了安全起见，最好这么做。因为这样的话，即使以后再增加新的 case 分支，也不用再在前面补充 break 语句了。

default 标签

如果没有任何一个 case 标签能匹配上 switch 表达式的值，程序将执行紧跟在 **default 标签**（default label）后面的语句。例如，可以增加一个计数值来统计非元音字母的数量，只要在 default 分支内不断递增名为 otherCnt 的变量就可以了：

```
// 如果 ch 是一个元音字母，将相应的计数值加 1
switch (ch) {
    case 'a': case 'e': case 'i': case 'o': case 'u':
        ++vowelCnt;
        break;
    default:
```

```
            ++otherCnt;
            break;
        }
    }
```

在这个版本的程序中，如果 ch 不是元音字母，就从 default 标签开始执行并把 otherCnt 加 1。 182



即使不准备在 default 标签下做任何工作，定义一个 default 标签也是有用的。其目的在于告诉程序的读者，我们已经考虑到了默认的情况，只是目前什么也没做。

标签不应该孤零零地出现，它后面必须跟上一条语句或者另外一个 case 标签。如果 switch 结构以一个空的 default 标签作为结束，则该 default 标签后面必须跟上一条空语句或一个空块。

switch 内部的变量定义

如前所述，switch 的执行流程有可能会跨过某些 case 标签。如果程序跳转到了某个特定的 case，则 switch 结构中该 case 标签之前的部分会被忽略掉。这种忽略掉一部分代码的行为引出了一个有趣的问题：如果被略过的代码中含有变量的定义该怎么办？

答案是：如果在某处一个带有初值的变量位于作用域之外，在另一处该变量位于作用域之内，则从前一处跳转到后一处的行为是非法行为。

```
case true:
    // 因为程序的执行流程可能绕开下面的初始化语句，所以该 switch 语句不合法
    string file_name;     // 错误：控制流绕过一个隐式初始化的变量
    int ival = 0;         // 错误：控制流绕过一个显式初始化的变量
    int jval;             // 正确：因为 jval 没有初始化
    break;
case false:
    // 正确：jval 虽然在作用域内，但是它没有被初始化
    jval = next_num();        // 正确：给 jval 赋一个值
    if (file_name.empty())    // file_name 在作用域内，但是没有被初始化
        // ...
```

假设上述代码合法，则一旦控制流直接跳到 false 分支，也就同时略过了变量 file_name 和 ival 的初始化过程。此时这两个变量位于作用域之内，跟在 false 之后的代码试图在尚未初始化的情况下使用它们，这显然是行不通的。因此 C++语言规定，不允许跨过变量的初始化语句直接跳转到该变量作用域内的另一个位置。

如果需要为某个 case 分支定义并初始化一个变量，我们应该把变量定义在块内，从而确保后面的所有 case 标签都在变量的作用域之外。

```
case true:
    {
        // 正确：声明语句位于语句块内部
        string file_name = get_file_name();
        // ...
    }
    break;
case false:
```

```
if (file_name.empty()) // 错误：file_name 不在作用域之内
```

183

5.3.2 节练习

练习 5.9：编写一段程序，使用一系列 if 语句统计从 cin 读入的文本中有多少元音字母。

练习 5.10：我们之前实现的统计元音字母的程序存在一个问题：如果元音字母以大写形式出现，不会被统计在内。编写一段程序，既统计元音字母的小写形式，也统计大写形式，也就是说，新程序遇到'a'和'A'都应该递增 aCnt 的值，以此类推。

练习 5.11：修改统计元音字母的程序，使其也能统计空格、制表符和换行符的数量。

练习 5.12：修改统计元音字母的程序，使其能统计以下含有两个字符的字符序列的数量：ff、fl 和 fi。

练习 5.13：下面显示的每个程序都含有一个常见的编程错误，指出错误在哪里，然后修改它们。

(a)
```
unsigned aCnt = 0, eCnt = 0, iouCnt = 0;
char ch = next_text();
switch (ch) {
    case 'a': aCnt++;
    case 'e': eCnt++;
    default: iouCnt++;
}
```

(b)
```
unsigned index = some_value();
switch (index) {
    case 1:
        int ix = get_value();
        ivec[ ix ] = index;
        break;
    default:
        ix = ivec.size()-1;
        ivec[ ix ] = index;
}
```

(c)
```
unsigned evenCnt = 0, oddCnt = 0;
int digit = get_num() % 10;
switch (digit) {
    case 1, 3, 5, 7, 9:
        oddcnt++;
        break;
    case 2, 4, 6, 8, 10:
        evencnt++;
        break;
}
```

(d)
```
unsigned ival=512, jval=1024, kval=4096;
unsigned bufsize;
unsigned swt = get_bufCnt();
switch(swt) {
```

```
        case ival:
            bufsize = ival * sizeof(int);
            break;
        case jval:
            bufsize = jval * sizeof(int);
            break;
        case kval:
            bufsize = kval * sizeof(int);
            break;
}
```

5.4 迭代语句

迭代语句通常称为循环，它重复执行操作直到满足某个条件才停下来。while 和 for 语句在执行循环体之前检查条件，do while 语句先执行循环体，然后再检查条件。

5.4.1 while 语句



只要条件为真，**while 语句**（while statement）就重复地执行循环体，它的语法形式是

```
while (condition)
    statement
```

在 while 结构中，只要 *condition* 的求值结果为真就一直执行 *statement*（常常是一个块）。*condition* 不能为空，如果 *condition* 第一次求值就得 false，*statement* 一次都不执行。

while 的条件部分可以是一个表达式或者是一个带初始化的变量声明（参见 5.2 节，第 155 页）。通常来说，应该由条件本身或者是循环体设法改变表达式的值，否则循环可能无法终止。



定义在 while 条件部分或者 while 循环体内的变量每次迭代都经历从创建到销毁的过程。

使用 while 循环

当不确定到底要迭代多少次时，使用 while 循环比较合适，比如读取输入的内容就是如此。还有一种情况也应该使用 while 循环，这就是我们想在循环结束后访问循环控制变量。例如：

185

```
vector<int> v;
int i;
// 重复读入数据，直至到达文件末尾或者遇到其他输入问题
while (cin >> i)
    v.push_back(i);
// 寻找第一个负值元素
auto beg = v.begin();
while (beg != v.end() && *beg >= 0)
    ++beg;
if (beg == v.end())
    // 此时我们知道 v 中的所有元素都大于等于 0
```

第一个循环从标准输入中读取数据，我们一开始不清楚循环要执行多少次，当 cin 读取到无效数据、遇到其他一些输入错误或是到达文件末尾时循环条件失效。第二个循环重复执行直到遇到一个负值为止，循环终止后，beg 或者等于 v.end()，或者指向 v 中一个小于 0 的元素。可以在 while 循环外继续使用 beg 的状态以进行其他处理。

5.4.1 节练习

练习 5.14：编写一段程序，从标准输入中读取若干 string 对象并查找连续重复出现的单词。所谓连续重复出现的意思是：一个单词后面紧跟着这个单词本身。要求记录连续重复出现的最大次数以及对应的单词。如果这样的单词存在，输出重复出现的最大次数；如果不存在，输出一条信息说明任何单词都没有连续出现过。例如，如果输入是

```
how now now now brown cow cow
```

那么输出应该表明单词 now 连续出现了 3 次。



5.4.2 传统的 for 语句

for 语句的语法形式是

```
for (init-statemen; condition; expression)
    statement
```

关键字 for 及括号里的部分称作 for 语句头。

init-statement 必须是以下三种形式中的一种：声明语句、表达式语句或者空语句，因为这些语句都以分号作为结束，所以 for 语句的语法形式也可以看做

```
for (initializer; condition; expression)
    statement
```

一般情况下，*init-statement* 负责初始化一个值，这个值将随着循环的进行而改变。*condition* 作为循环控制的条件，只要 *condition* 为真，就执行一次 *statement*。如果 *condition* 第一次的求值结果就是 false，则 *statement* 一次也不会执行。*expression* 负责修改 *init-statement* 初始化的变量，这个变量正好就是 *condition* 检查的对象，修改发生在每次循环迭代之后。*statement* 可以是一条单独的语句也可以是一条复合语句。

186

传统 for 循环的执行流程

我们以 3.2.3 节（第 85 页）的 for 循环为例：

```
// 重复处理 s 中的字符直至我们处理完全部字符或者遇到了一个表示空白的字符
for (decltype(s.size()) index = 0;
     index != s.size() && !isspace(s[index]); ++index)
         s[index] = toupper(s[index]); // 将当前字符改成大写形式
```

求值的顺序如下所示：

1. 循环开始时，首先执行一次 *init-statement*。此例中，定义 index 并初始化为 0。
2. 接下来判断 *condition*。如果 index 不等于 s.size() 而且在 s[index] 位置的字符不是空白，则执行 for 循环体的内容。否则，循环终止。如果第一次迭代时条件就为假，for 循环体一次也不会执行。
3. 如果条件为真，执行循环体。此例中，for 循环体将 s[index] 位置的字符改写

成大写形式。

4. 最后执行 *expression*。此例中，将 index 的值加 1。

这 4 步说明了 for 循环第一次迭代的过程。其中第 1 步只在循环开始时执行一次，第 2、3、4 步重复执行直到条件为假时终止，也就是在 s 中遇到一个空白字符或者 index 大于 s.size() 时终止。



牢记 for 语句头中定义的对象只在 for 循环体内可见。因此在上面的例子中，for 循环结束后 index 就不可用了。

for 语句头中的多重定义

和其他的声明一样，*init-statement* 也可以定义多个对象。但是 *init-statement* 只能有一条声明语句，因此，所有变量的基础类型必须相同（参见 2.3 节，第 45 页）。举个例子，我们用下面的循环把 vector 的元素拷贝一份添加到原来的元素后面：

```
// 记录下 v 的大小，当到达原来的最后一个元素后结束循环
for (decltype(v.size()) i = 0, sz = v.size(); i != sz; ++i)
    v.push_back(v[i]);
```

在这个循环中，我们在 *init-statement* 里同时定义了索引 i 和循环控制变量 sz。

省略 for 语句头的某些部分

187

for 语句头能省略掉 *init-statement*、*condition* 和 *expression* 中的任何一个（或者全部）。

如果无须初始化，则我们可以使用一条空语句作为 *init-statement*。例如，对于在 vector 对象中寻找第一个负数的程序，完全能用 for 循环改写：

```
auto beg = v.begin();
for ( /* 空语句 */; beg != v.end() && *beg >= 0; ++beg)
    ; // 什么也不做
```

注意，分号必须保留以表明我们省略掉了 *init-statement*。说得更准确一点，分号表示的是一个空的 *init-statement*。在这个循环中，因为所有要做的工作都在 for 语句头的条件和表达式部分完成了，所以 for 循环体也是空的。其中，条件部分决定何时停止查找，表达式部分递增迭代器。

省略 *condition* 的效果等价于在条件部分写了一个 true。因为条件的值永远是 true，所以在循环体内必须有语句负责退出循环，否则循环就会无休止地执行下去：

```
for (int i = 0; /* 条件为空 */ ; ++i) {
    // 对 i 进行处理，循环内部的代码必须负责终止迭代过程！
}
```

我们也能省略掉 for 语句头中的 *expression*，但是在这样的循环中就要求条件部分或者循环体必须改变迭代变量的值。举个例子，之前有一个将整数读入 vector 的 while 循环，我们使用 for 语句改写它：

```
vector<int> v;
for (int i; cin >> i; /* 表达式为空 */ )
    v.push_back(i);
```

因为条件部分能改变 i 的值，所以这个循环无须表达式部分。其中，条件部分不断检查输

入流的内容，只要读取完所有的输入或者遇到一个输入错误就终止循环。

188

5.4.2 节练习

练习 5.15：说明下列循环的含义并改正其中的错误。

```
(a) for (int ix = 0; ix != sz; ++ix) { /* ...*/ }
    if (ix != sz)
        // ...
(b) int ix;
    for (ix != sz; ++ix) { /* ...*/ }
(c) for (int ix = 0; ix != sz; ++ix, ++ sz) { /* ...*/ }
```

练习 5.16：while 循环特别适用于那种条件保持不变、反复执行操作的情况，例如，当未达到文件末尾时不断读取下一个值。for 循环则更像是在按步骤迭代，它的索引值在某个范围内依次变化。根据每种循环的习惯用法各自编写一段程序，然后分别用另一种循环改写。如果只能使用一种循环，你倾向于使用哪种呢？为什么？

练习 5.17：假设有两个包含整数的 vector 对象，编写一段程序，检验其中一个 vector 对象是否是另一个的前缀。为了实现这一目标，对于两个不等长的 vector 对象，只需挑出长度较短的那个，把它的所有元素和另一个 vector 对象比较即可。例如，如果两个 vector 对象的元素分别是 0、1、1、2 和 0、1、1、2、3、5、8，则程序的返回结果应该为真。



5.4.3 范围 for 语句

C++11

C++11 新标准引入了一种更简单的 for 语句，这种语句可以遍历容器或其他序列的所有元素。**范围 for 语句**（range for statement）的语法形式是：

```
for (declaration : expression)
    statement
```

expression 表示的必须是一个序列，比如用花括号括起来的初始值列表（参见 3.3.1 节，第 88 页）、数组（参见 3.5 节，第 101 页）或者 vector 或 string 等类型的对象，这些类型的共同特点是拥有能返回迭代器的 begin 和 end 成员（参见 3.4 节，第 95 页）。

declaration 定义一个变量，序列中的每个元素都得能转换成该变量的类型（参见 4.11 节，第 141 页）。确保类型相容最简单的办法是使用 auto 类型说明符（参见 2.5.2 节，第 61 页），这个关键字可以令编译器帮助我们指定合适的类型。如果需要对序列中的元素执行写操作，循环变量必须声明成引用类型。

每次迭代都会重新定义循环控制变量，并将其初始化成序列中的下一个值，之后才会执行 *statement*。像往常一样，*statement* 可以是一条单独的语句也可以是一个块。所有元素都处理完毕后循环终止。

之前我们已经接触过几个这样的循环。接下来的例子将把 vector 对象中的每个元素都翻倍，它涵盖了范围 for 语句的几乎所有语法特征：

```
vector<int> v = {0,1,2,3,4,5,6,7,8,9};
// 范围变量必须是引用类型，这样才能对元素执行写操作
for (auto &r : v)     // 对于 v 中的每一个元素
```

```
    r *= 2;          // 将 v 中每个元素的值翻倍
```

for 语句头声明了循环控制变量 r，并把它和 v 关联在一起，我们使用关键字 auto 令编译器为 r 指定正确的类型。由于准备修改 v 的元素的值，因此将 r 声明成引用类型。此时，在循环体内给 r 赋值，即改变了 r 所绑定的元素的值。

范围 for 语句的定义来源于与之等价的传统 for 语句：

```
for (auto beg = v.begin(), end = v.end(); beg != end; ++beg) {
    auto &r = *beg;  // r 必须是引用类型，这样才能对元素执行写操作
    r *= 2;          // 将 v 中每个元素的值翻倍
}
```

学习了范围 for 语句的原理之后，我们也就不难理解为什么在 3.3.2 节（第 90 页）强调不能通过范围 for 语句增加 vector 对象（或者其他容器）的元素了。在范围 for 语句中，预存了 end() 的值。一旦在序列中添加（删除）元素，end 函数的值就可能变得无效了（参见 3.4.1 节，第 98 页）。关于这一点，将在 9.3.6 节（第 315 页）做更详细的介绍。 189

5.4.4 do while 语句

do while 语句（do while statement）和 while 语句非常相似，唯一的区别是，do while 语句先执行循环体后检查条件。不管条件的值如何，我们都至少执行一次循环。do while 语句的语法形式如下所示：

```
do
    statement
while (condition);
```



do while 语句应该在括号包围起来的条件后面用一个分号表示语句结束。

在 do 语句中，求 *condition* 的值之前首先执行一次 *statement*，*condition* 不能为空。如果 *condition* 的值为假，循环终止；否则，重复循环过程。*condition* 使用的变量必须定义在循环体之外。

我们可以使用 do while 循环（不断地）执行加法运算：

```
// 不断提示用户输入一对数，然后求其和
string rsp; // 作为循环的条件，不能定义在 do 的内部
do {
    cout << "please enter two values: ";
    int val1 = 0, val2 = 0;
    cin >> val1 >> val2;
    cout << "The sum of " << val1 << " and " << val2
         << " = " << val1 + val2 << "\n\n"
         << "More? Enter yes or no: ";
    cin  >> rsp;
} while (!rsp.empty() && rsp[0] != 'n');
```

循环首先提示用户输入两个数字，然后输出它们的和并询问用户是否继续。条件部分检查用户做出的回答，如果用户没有回答，或者用户的回答以字母 n 开始，循环都将终止。否则循环继续执行。

因为对于 do while 来说先执行语句或者块，后判断条件，所以不允许在条件部分

定义变量：

```
do {
    // ...
    mumble(foo);
} while (int foo = get_foo()); // 错误：将变量声明放在了 do 的条件部分
```

如果允许在条件部分定义变量，则变量的使用出现在定义之前，这显然是不合常理的！

190

5.4.4 节练习

练习 5.18：说明下列循环的含义并改正其中的错误。

```
(a) do
        int v1, v2;
        cout << "Please enter two numbers to sum:" ;
        if (cin >> v1 >> v2)
            cout << "Sum is: " << v1 + v2 << endl;
    while (cin);
(b) do {
        // ...
    } while (int ival = get_response());
(c) do {
        int ival = get_response();
    } while (ival);
```

练习 5.19：编写一段程序，使用 do while 循环重复地执行下述任务：首先提示用户输入两个 string 对象，然后挑出较短的那个并输出它。

5.5 跳转语句

跳转语句中断当前的执行过程。C++语言提供了 4 种跳转语句：break、continue、goto 和 return。本章介绍前三种跳转语句，return 语句将在 6.3 节（第 199 页）进行介绍。

5.5.1 break 语句

break 语句（break statement）负责终止离它最近的 while、do while、for 或 switch 语句，并从这些语句之后的第一条语句开始继续执行。

break 语句只能出现在迭代语句或者 switch 语句内部（包括嵌套在此类循环里的语句或块的内部）。break 语句的作用范围仅限于最近的循环或者 switch：

```
string buf;
while (cin >> buf && !buf.empty()) {
    switch(buf[0]) {
    case '-':
        // 处理到第一个空白为止
        for (auto it = buf.begin()+1; it != buf.end(); ++it) {
            if (*it == ' ')
                break; // #1，离开 for 循环
            // ...
```

```
            }
            // break #1 将控制权转移到这里
            // 剩余的'-'处理:
            break; // #2, 离开 switch 语句
            case '+':
                //...
        } // 结束 switch
        // 结束 switch: break #2 将控制权转移到这里
    } // 结束 while
```

191

标记为#1 的 break 语句负责终止连字符 case 标签后面的 for 循环。它不但不会终止 switch 语句，甚至连当前的 case 分支也终止不了。接下来，程序继续执行 for 循环之后的第一条语句，这条语句可能接着处理连字符的情况，也可能是另一条用于终止当前分支的 break 语句。

标记为#2 的 break 语句负责终止 switch 语句，但是不能终止 while 循环。执行完这个 break 后，程序继续执行 while 的条件部分。

5.5.1 节练习

练习 5.20：编写一段程序，从标准输入中读取 string 对象的序列直到连续出现两个相同的单词或者所有单词都读完为止。使用 while 循环一次读取一个单词，当一个单词连续出现两次时使用 break 语句终止循环。输出连续重复出现的单词，或者输出一个消息说明没有任何单词是连续重复出现的。

5.5.2 continue 语句

continue 语句（continue statement）终止最近的循环中的当前迭代并立即开始下一次迭代。continue 语句只能出现在 for、while 和 do while 循环的内部，或者嵌套在此类循环里的语句或块的内部。和 break 语句类似的是，出现在嵌套循环中的 continue 语句也仅作用于离它最近的循环。和 break 语句不同的是，只有当 switch 语句嵌套在迭代语句内部时，才能在 switch 里使用 continue。

continue 语句中断当前的迭代，但是仍然继续执行循环。对于 while 或者 do while 语句来说，继续判断条件的值；对于传统的 for 循环来说，继续执行 for 语句头的 *expression*；而对于范围 *for* 语句来说，则是用序列中的下一个元素初始化循环控制变量。

例如，下面的程序每次从标准输入中读取一个单词。循环只对那些以下画线开头的单词感兴趣，其他情况下，我们直接终止当前的迭代并获取下一个单词：

```
string buf;
while (cin >> buf && !buf.empty()) {
    if (buf[0] != '_')
        continue; // 接着读取下一个输入
    // 程序执行过程到了这里？说明当前的输入是以下画线开始的；接着处理 buf……
}
```

5.5.2 节练习

192

练习 5.21：修改 5.5.1 节（第 171 页）练习题的程序，使其找到的重复单词必须以大写字母开头。

5.5.3 goto 语句

goto 语句（goto statement）的作用是从 goto 语句无条件跳转到同一函数内的另一条语句。

Best Practices 不要在程序中使用 goto 语句，因为它使得程序既难理解又难修改。

goto 语句的语法形式是

```
goto label;
```

其中，*label* 是用于标识一条语句的标示符。**带标签语句**（labeled statement）是一种特殊的语句，在它之前有一个标示符以及一个冒号：

```
end: return; // 带标签语句，可以作为 goto 的目标
```

标签标示符独立于变量或其他标示符的名字，因此，标签标示符可以和程序中其他实体的标示符使用同一个名字而不会相互干扰。goto 语句和控制权转向的那条带标签的语句必须位于同一个函数之内。

和 switch 语句类似，goto 语句也不能将程序的控制权从变量的作用域之外转移到作用域之内：

```
    // ...
    goto end;
    int ix = 10; // 错误：goto 语句绕过了一个带初始化的变量定义
end:
    // 错误：此处的代码需要使用 ix，但是 goto 语句绕过了它的声明
    ix = 42;
```

向后跳过一个已经执行的定义是合法的。跳回到变量定义之前意味着系统将销毁该变量，然后重新创建它：

```
// 向后跳过一个带初始化的变量定义是合法的
begin:
    int sz = get_size();
    if (sz <= 0) {
        goto begin;
    }
```

在上面的代码中，goto 语句执行后将销毁 sz。因为跳回到 begin 的动作跨过了 sz 的定义语句，所以 sz 将重新定义并初始化。

193

5.5.3 节练习

练习 5.22：本节的最后一个例子跳回到 begin，其实使用循环能更好地完成该任务。重写这段代码，注意不再使用 goto 语句。

5.6 try 语句块和异常处理

异常是指存在于运行时的反常行为，这些行为超出了函数正常功能的范围。典型的异常包括失去数据库连接以及遇到意外输入等。处理反常行为可能是设计所有系统最难的一部分。

当程序的某部分检测到一个它无法处理的问题时，需要用到异常处理。此时，检测出问题的部分应该发出某种信号以表明程序遇到了故障，无法继续下去了，而且信号的发出方无须知道故障将在何处得到解决。一旦发出异常信号，检测出问题的部分也就完成了任务。

如果程序中含有可能引发异常的代码，那么通常也会有专门的代码处理问题。例如，如果程序的问题是输入无效，则异常处理部分可能会要求用户重新输入正确的数据；如果丢失了数据库连接，会发出报警信息。

异常处理机制为程序中异常检测和异常处理这两部分的协作提供支持。在 C++语言中，异常处理包括：

- **throw 表达式**（throw expression），异常检测部分使用 throw 表达式来表示它遇到了无法处理的问题。我们说 throw **引发**（raise）了异常。
- **try 语句块**（try block），异常处理部分使用 try 语句块处理异常。try 语句块以关键字 try 开始，并以一个或多个 **catch 子句**（catch clause）结束。try 语句块中代码抛出的异常通常会被某个 catch 子句处理。因为 catch 子句“处理”异常，所以它们也被称作**异常处理代码**（exception handler）。
- 一套**异常类**（exception class），用于在 throw 表达式和相关的 catch 子句之间传递异常的具体信息。

在本节的剩余部分，我们将分别介绍异常处理的这三个组成部分。在 18.1 节（第 684 页）还将介绍更多关于异常的知识。

5.6.1 throw 表达式

程序的异常检测部分使用 throw 表达式引发一个异常。throw 表达式包含关键字 throw 和紧随其后的一个表达式，其中表达式的类型就是抛出的异常类型。throw 表达式后面通常紧跟一个分号，从而构成一条表达式语句。

举个简单的例子，回忆 1.5.2 节（第 20 页）把两个 Sales_item 对象相加的程序。194 这个程序检查它读入的记录是否是关于同一种书籍的，如果不是，输出一条信息然后退出。

```
Sales_item item1, item2;
cin >> item1 >> item2;
// 首先检查 item1 和 item2 是否表示同一种书籍
if (item1.isbn() == item2.isbn()) {
    cout << item1 + item2 << endl;
    return 0; // 表示成功
} else {
    cerr << "Data must refer to same ISBN" << endl;
    return -1; // 表示失败
}
```

在真实的程序中，应该把对象相加的代码和用户交互的代码分离开来。此例中，我们改写程序使得检查完成后不再直接输出一条信息，而是抛出一个异常：

```
// 首先检查两条数据是否是关于同一种书籍的
if (item1.isbn() != item2.isbn())
    throw runtime_error("Data must refer to same ISBN");
// 如果程序执行到了这里，表示两个 ISBN 是相同的
cout << item1 + item2 << endl;
```

在这段代码中，如果 ISBN 不一样就抛出一个异常，该异常是类型 runtime_error 的对象。抛出异常将终止当前的函数，并把控制权转移给能处理该异常的代码。

类型 runtime_error 是标准库异常类型的一种，定义在 stdexcept 头文件中。关于标准库异常类型更多的知识将在 5.6.3 节（第 176 页）介绍。我们必须初始化 runtime_error 的对象，方式是给它提供一个 string 对象或者一个 C 风格的字符串（参见 3.5.4 节，第 109 页），这个字符串中有一些关于异常的辅助信息。

5.6.2 try 语句块

try 语句块的通用语法形式是

```
try {
    program-statements
} catch (exception-declaration) {
    handler-statements
} catch (exception-declaration) {
    handler-statements
} // ...
```

try 语句块的一开始是关键字 try，随后紧跟着一个块，这个块就像大多数时候那样是花括号括起来的语句序列。

195 跟在 try 块之后的是一个或多个 catch 子句。catch 子句包括三部分：关键字 catch、括号内一个（可能未命名的）对象的声明（称作**异常声明**，exception declaration）以及一个块。当选中了某个 catch 子句处理异常之后，执行与之对应的块。catch 一旦完成，程序跳转到 try 语句块最后一个 catch 子句之后的那条语句继续执行。

try 语句块中的 *program-statements* 组成程序的正常逻辑，像其他任何块一样，*program-statements* 可以有包括声明在内的任意 C++语句。一如往常，try 语句块内声明的变量在块外部无法访问，特别是在 catch 子句内也无法访问。

编写处理代码

在之前的例子里，我们使用了一个 throw 表达式以避免把两个代表不同书籍的 Sales_item 相加。我们假设执行 Sales_item 对象加法的代码是与用户交互的代码分离开来的。其中与用户交互的代码负责处理发生的异常，它的形式可能如下所示：

```
while (cin >> item1 >> item2) {
    try {
        // 执行添加两个 Sales_item 对象的代码
        // 如果添加失败，代码抛出一个 runtime_error 异常
    } catch (runtime_error err) {
        // 提醒用户两个 ISBN 必须一致，询问是否重新输入
        cout << err.what()
             << "\nTry Again? Enter y or n" << endl;
        char c;
        cin >> c;
```