

本例中声明了一个没有参数的 `const` 版本的 `sorted`，此声明是错误的。因为 `Foo` 类中还有一个无参的 `sorted` 版本，它有一个引用限定符，因此 `const` 版本也必须有引用限定符。另一方面，接受一个比较操作指针的 `sorted` 版本是没问题的，因为两个函数都没有引用限定符。



如果一个成员函数有引用限定符，则具有相同参数列表的所有版本都必须有引用限定符。

13.6.3 节练习

练习 13.55: 为你的 `StrBlob` 添加一个右值引用版本的 `push_back`。

练习 13.56: 如果 `sorted` 定义如下，会发生什么：

```
Foo Foo::sorted() const & {  
    Foo ret(*this);  
    return ret.sorted();  
}
```

练习 13.57: 如果 `sorted` 定义如下，会发生什么：

```
Foo Foo::sorted() const & { return Foo(*this).sorted(); }
```

练习 13.58: 编写新版本的 `Foo` 类，其 `sorted` 函数中有打印语句，测试这个类，来验证你对前两题的答案是否正确。

done!
done!
done!

549 小结

每个类都会控制该类型对象拷贝、移动、赋值以及销毁时发生什么。特殊的成员函数——拷贝构造函数、移动构造函数、拷贝赋值运算符、移动赋值运算符和析构函数定义了这些操作。移动构造函数和移动赋值运算符接受一个（通常是非 `const` 的）右值引用；而拷贝版本则接受一个（通常是 `const` 的）普通左值引用。

如果一个类未声明这些操作，编译器会自动为其生成。如果这些操作未定义成删除的，它们会逐成员初始化、移动、赋值或销毁对象：合成的操作依次处理每个非 `static` 数据成员，根据成员类型确定如何移动、拷贝、赋值或销毁它。

分配了内存或其他资源的类几乎总是需要定义拷贝控制成员来管理分配的资源。如果一个类需要析构函数，则它几乎肯定也需要定义移动和拷贝构造函数及移动和拷贝赋值运算符。

术语表

拷贝并交换 (copy and swap) 涉及赋值运算符的技术，首先拷贝右侧运算对象，然后调用 `swap` 来交换副本和左侧运算对象。

拷贝赋值运算符 (copy-assignment operator) 接受一个本类型对象的赋值运算符版本。通常，拷贝赋值运算符的参数是一个 `const` 的引用，并返回指向本对象的引用。如果类未显式定义拷贝赋值运算符，编译器会为它合成一个。

拷贝构造函数 (copy constructor) 一种构造函数，将新对象初始化为同类型另一个对象的副本。当向函数传递对象，或以传值方式从函数返回对象时，会隐式使用拷贝构造函数。如果我们未提供拷贝构造函数，编译器会为我们合成一个。

拷贝控制 (copy control) 特殊的成员函数，控制拷贝、移动、赋值及销毁本类类型对象时发生什么。如果类未定义这些操作，编译器会为它合成恰当的定义。

拷贝初始化 (copy initialization) 一种初始化形式，当我们使用 `=` 为一个新创建的对象提供初始化器时，会使用拷贝初始化。如果我们向函数传递对象或以传值方式从函数返回对象，以及初始化一个数组或一个聚合类时，也会使用拷贝初始化。

删除的函数 (deleted function) 不能使用的函数。我们在一个函数的声明上指定 `=delete` 来删除它。删除的函数的一个常见用途是告诉编译器不要为类合成拷贝和/或移动操作。

析构函数 (destructor) 特殊的成员函数，当对象离开作用域或被释放时进行清理工作。编译器会自动销毁每个数据成员。类类型的成员通过调用其析构函数来销毁；而内置类型或复合类型的成员的销毁则不需要做任何工作。特别是，析构函数不会释放指针成员指向的对象。

左值引用 (lvalue reference) 可以绑定到左值的引用。

550 逐成员拷贝/赋值 (memberwise copy/assign) 合成的拷贝与移动构造函数及拷贝与移动赋值运算符的工作方式。合成的拷贝或移动构造函数依次处理每个非 `static` 数据成员，通过从给定对象拷贝或移动对应成员来初始化本对象成员；拷贝或移动赋值运算符从右侧运算对象中将每个成员拷贝赋值或移动赋值到左侧运算对象中。内置类型或复合类型的成员直接进行初始化或赋值。类类型的成员通过成员对应的拷贝/移动构造函数或拷贝/移动赋值运算符进行初始化或赋值。

move 用来将一个右值引用绑定到一个左值的标准库函数。调用 **move** 隐含地承诺我们将不会再使用移后源对象，除了销毁它或赋予它一个新值之外。

移动赋值运算符 (**move-assignment operator**) 接受一个本类型右值引用参数的赋值运算符版本。通常，移动赋值运算符将数据从右侧运算对象移动到左侧运算对象。赋值之后，对右侧运算对象执行析构函数必须是安全的。

移动构造函数 (**move constructor**) 一种构造函数，接受一个本类型的右值引用。通常，移动构造函数将数据从其参数移动到新创建的对象中。移动之后，对给定的实参执行析构函数必须是安全的。

移动迭代器 (**move iterator**) 迭代器适配器，它生成的迭代器在解引用时会得到一个右值引用。

重载运算符 (**overloaded operator**) 一种函数，重定义了运算符应用于类类型的对象时的含义。本章介绍了如何定义赋值运算符；第 14 章中将介绍重载运算符的更多细节内容。

引用计数 (**reference count**) 一种程序设计技术，通常用于拷贝控制成员的设计。引用计数记录了有多少对象共享状态。构造函数（不是拷贝/移动构造函数）将引用计数置为 1。每当创建一个新副本时，计数

值递增。当一个对象被销毁时，计数值递减。赋值运算符和析构函数检查递减的引用计数是否为 0，如果是，它们会销毁对象。

引用限定符 (**reference qualifier**) 用来指出一个非 **static** 成员函数可以用于左值或右值的符号。限定符 **&** 和 **&&** 应该放在参数列表之后或 **const** 限定符之后（如果有的话）。被 **&** 限定的函数只能用于左值；被 **&&** 限定的函数只能用于右值。

右值引用 (**rvalue reference**) 指向一个将要销毁的对象的引用。

合成赋值运算符 (**synthesized assignment operator**) 编译器为未显式定义赋值运算符的类创建的（合成的）拷贝或移动赋值运算符版本。除非定义为删除的，合成赋值运算符会逐成员地将右侧运算对象赋予（移动到）左侧运算对象。

合成拷贝/移动构造函数 (**synthesized copy/move constructor**) 编译器为未显式定义对应的构造函数的类生成的拷贝或移动构造函数版本。除非定义为删除的，合成拷贝或移动构造函数分别通过从给定对象拷贝或移动成员来逐成员地初始化新对象。

合成析构函数 (**synthesized destructor**) 编译器为未显式定义析构函数的类创建的（合成的）版本。合成析构函数的函数体为空。

done!

done

done

dore!

dore

第 14 章

重载运算与类型转换

内容

14.1 基本概念.....	490
14.2 输入和输出运算符.....	494
14.3 算术和关系运算符.....	497
14.4 赋值运算符.....	499
14.5 下标运算符.....	501
14.6 递增和递减运算符.....	502
14.7 成员访问运算符.....	504
14.8 函数调用运算符.....	506
14.9 重载、类型转换与运算符.....	514
小结.....	523
术语表.....	523

在第 4 章中我们看到，C++ 语言定义了大量运算符以及内置类型的自动转换规则。这些特性使得程序员能编写出形式丰富、含有多种混合类型的表达式。

当运算符被用于类类型的对象时，C++ 语言允许我们为其指定新的含义；同时，我们也能自定义类类型之间的转换规则。和内置类型的转换一样，类类型转换隐式地将一种类型的对象转换成另一种我们所需类型的对象。

552

当运算符作用于类类型的运算对象时，可以通过运算符重载重新定义该运算符的含义。明智地使用运算符重载能令我们的程序更易于编写和阅读。举个例子，因为在 `Sales_item` 类（参见 1.5.1 节，第 17 页）中定义了输入、输出和加法运算符，所以可以通过下述形式输出两个 `Sales_item` 的和：

```
cout << item1 + item2;           // 输出两个 Sales_item 的和
```

相反的，由于我们的 `Sales_data` 类（参见 7.1 节，第 228 页）还没有重载这些运算符，因此它的加法代码显得比较冗长而不清晰：

```
print(cout, add(data1, data2)); // 输出两个 Sales_data 的和
```



14.1 基本概念

重载的运算符是具有特殊名字的函数：它们的名字由关键字 `operator` 和其后要定义的运算符号共同组成。和其他函数一样，重载的运算符也包含返回类型、参数列表以及函数体。

重载运算符函数的参数数量与该运算符作用的运算对象数量一样多。一元运算符有一个参数，二元运算符有两个。对于二元运算符来说，左侧运算对象传递给第一个参数，而右侧运算对象传递给第二个参数。除了重载的函数调用运算符 `operator()` 之外，其他重载运算符不能含有默认实参（参见 6.5.1 节，第 211 页）。

如果一个运算符函数是成员函数，则它的第一个（左侧）运算对象绑定到隐式的 `this` 指针上（参见 7.1.2 节，第 231 页），因此，成员运算符函数的（显式）参数数量比运算符的运算对象总数少一个。

Note

当一个重载的运算符是成员函数时，`this` 绑定到左侧运算对象。成员运算符函数的（显式）参数数量比运算对象的数量少一个。

对于一个运算符函数来说，它或者是类的成员，或者至少含有一个类类型的参数：

```
// 错误：不能为 int 重定义内置的运算符
int operator+(int, int);
```

这一约定意味着当运算符作用于内置类型的运算对象时，我们无法改变该运算符的含义。

我们可以重载大多数（但不是全部）运算符。表 14.1 指明了哪些运算符可以被重载，哪些不行。我们将在 19.1.1 节（第 726 页）介绍重载 `new` 和 `delete` 的方法。

我们只能重载已有的运算符，而无权发明新的运算符号。例如，我们不能提供 `operator**` 来执行幂操作。

有四个符号（`+`、`-`、`*`、`&`）既是一元运算符也是二元运算符，所有这些运算符都能被重载，从参数的数量我们可以推断到底定义的是哪种运算符。

553

对于一个重载的运算符来说，其优先级和结合律（参见 4.1.2 节，第 121 页）与对应的内置运算符保持一致。不考虑运算对象类型的话，

```
x == y + z;
```

永远等价于 `x == (y + z)`。

表 14.1: 运算符

可以被重载的运算符					
+	-	*	/	%	^
&		~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	
+=	-=	/=	%=	^=	&=
=	*=	<<=	>>=	[]	()
->	->*	new	new[]	delete	delete[]
不能被重载的运算符					
::	.*	.	?:		

直接调用一个重载的运算符函数

通常情况下，我们将运算符作用于类型正确的实参，从而以这种间接方式“调用”重载的运算符函数。然而，我们也能像调用普通函数一样直接调用运算符函数，先指定函数名字，然后传入数量正确、类型适当的实参：

```
// 一个非成员运算符函数的等价调用
data1 + data2;           // 普通的表达式
operator+(data1, data2); // 等价的函数调用
```

这两次调用是等价的，它们都调用了非成员函数 `operator+`，传入 `data1` 作为第一个实参、传入 `data2` 作为第二个实参。

我们像调用其他成员函数一样显式地调用成员运算符函数。具体做法是，首先指定运行函数的对象（或指针）的名字，然后使用点运算符（或箭头运算符）访问希望调用的函数：

```
data1 += data2;           // 基于“调用”的表达式
data1.operator+=(data2); // 对成员运算符函数的等价调用
```

这两条语句都调用了成员函数 `operator+=`，将 `this` 绑定到 `data1` 的地址、将 `data2` 作为实参传入了函数。

某些运算符不应该被重载

回忆之前介绍过的，某些运算符指定了运算对象求值的顺序。因为使用重载的运算符本质上是一次函数调用，所以这些关于运算对象求值顺序的规则无法应用到重载的运算符上。特别是，逻辑与运算符、逻辑或运算符（参见 4.3 节，第 126 页）和逗号运算符（参见 4.10 节，第 140 页）的运算对象求值顺序规则无法保留下来。除此之外，`&&` 和 `||` 运算符的重载版本也无法保留内置运算符的短路求值属性，两个运算对象总是会被求值。

因为上述运算符的重载版本无法保留求值顺序和/或短路求值属性，因此不建议重载它们。当代码使用了这些运算符的重载版本时，用户可能会突然发现他们一直习惯的求值规则不再适用了。

还有一个原因使得我们一般不重载逗号运算符和取地址运算符：C++ 语言已经定义了这两种运算符用于类类型对象时的特殊含义，这一点与大多数运算符都不相同。因为这两种运算符已经有了内置的含义，所以一般来说它们不应该被重载，否则它们的行为将异于常态，从而导致类的用户无法适应。

Best
Practices

通常情况下，不应该重载逗号、取地址、逻辑与和逻辑或运算符。

使用与内置类型一致的含义

当你开始设计一个类时，首先应该考虑的是这个类将提供哪些操作。在确定类需要哪些操作之后，才能思考到底应该把每个类操作设成普通函数还是重载的运算符。如果某些操作在逻辑上与运算符相关，则它们适合于定义成重载的运算符：

- 如果类执行 IO 操作，则定义移位运算符使其与内置类型的 IO 保持一致。
- 如果类的某个操作是检查相等性，则定义 `operator==`；如果类有了 `operator==`，意味着它通常也应该有 `operator!=`。
- 如果类包含一个内在的单序比较操作，则定义 `operator<`；如果类有了 `operator<`，则它也应该含有其他关系操作。
- 重载运算符的返回类型通常情况下应该与其内置版本的返回类型兼容：逻辑运算符和关系运算符应该返回 `bool`，算术运算符应该返回一个类类型的值，赋值运算符和复合赋值运算符则应该返回左侧运算对象的一个引用。

提示：尽量明智地使用运算符重载

每个运算符在用于内置类型时都有比较明确的含义。以二元+运算符为例，它明显执行的是加法操作。因此，把二元+运算符映射到类类型的一个类似操作上可以极大地简化记忆。例如对于标准库类型 `string` 来说，我们就会使用+把一个 `string` 对象连接到另一个后面，很多编程语言都有类似的用法。

当在内置的运算符和我们自己的操作之间存在逻辑映射关系时，运算符重载的效果最好。此时，使用重载的运算符显然比另起一个名字更自然也更直观。不过，过分滥用运算符重载也会使我们的类变得难以理解。

在实际编程过程中，一般没有特别明显的滥用运算符重载的情况。例如，一般来说没有哪个程序员会定义 `operator+` 并让它执行减法操作。然而经常发生的一种情况是，程序员可能会强行扭曲了运算符的“常规”含义使得其适应某种给定的类型，这显然是我们不希望发生的。因此我们的建议是：只有当操作的含义对于用户来说清晰明了时才使用运算符。如果用户对运算符可能有几种不同的理解，则使用这样的运算符将产生二义性。

赋值和复合赋值运算符

赋值运算符的行为与复合版本的类似：赋值之后，左侧运算对象和右侧运算对象的值相等，并且运算符应该返回它左侧运算对象的一个引用。重载的赋值运算应该继承而非违背其内置版本的含义。

555

如果类含有算术运算符（参见 4.2 节，第 124 页）或者位运算符（参见 4.8 节，第 136 页），则最好也提供对应的复合赋值运算符。无须赘言，`+=` 运算符的行为显然应该与其内置版本一致，即先执行+，再执行=。

选择作为成员或者非成员

当我们定义重载的运算符时，必须首先决定是将其声明为类的成员函数还是声明为一个普通的非成员函数。在某些时候我们别无选择，因为有的运算符必须作为成员；另一些

情况下，运算符作为普通函数比作为成员更好。

下面的准则有助于我们在将运算符定义为成员函数还是普通的非成员函数做出抉择：

- 赋值(=)、下标([])、调用(())和成员访问箭头(->)运算符必须是成员。
- 复合赋值运算符一般来说应该是成员，但并非必须，这一点与赋值运算符略有不同。
- 改变对象状态的运算符或者与给定类型密切相关的运算符，如递增、递减和解引用运算符，通常应该是成员。
- 具有对称性的运算符可能转换任意一端的运算对象，例如算术、相等性、关系和位运算符等，因此它们通常应该是普通的非成员函数。

程序员希望能在含有混合类型的表达式中使用对称性运算符。例如，我们能求一个 int 和一个 double 的和，因为它们中的任意一个都可以是左侧运算对象或右侧运算对象，所以加法是对称的。如果我们想提供含有类对象的混合类型表达式，则运算符必须定义成非成员函数。

556

当我们把运算符定义成成员函数时，它的左侧运算对象必须是运算符所属类的一个对象。例如：

```
string s = "world";
string t = s + "!"; // 正确：我们能将一个 const char* 加到一个 string 对象上
string u = "hi" + s; // 如果 + 是 string 的成员，则产生错误
```

如果 operator+ 是 string 类的成员，则上面的第一个加法等价于 s.operator+("!")。同样的，"hi"+s 等价于 "hi".operator+(s)。显然 "hi" 的类型是 const char*，这是一种内置类型，根本就没有成员函数。

因为 string 将 + 定义成了普通的非成员函数，所以 "hi"+s 等价于 operator+("hi", s)。和任何其他函数调用一样，每个实参都能被转换成形参类型。唯一的要求是至少有一个运算对象是类类型，并且两个运算对象都能准确无误地转换成 string。

14.1 节练习

练习 14.1: 在什么情况下重载的运算符与内置运算符有所区别？在什么情况下重载的运算符又与内置运算符一样？

练习 14.2: 为 Sales_data 编写重载的输入、输出、加法和复合赋值运算符。

练习 14.3: string 和 vector 都定义了重载的==以比较各自的对象，假设 svec1 和 svec2 是存放 string 的 vector，确定在下面的表达式中分别使用了哪个版本的==？

- (a) "cobble" == "stone" (b) svec1[0] == svec2[0]
(c) svec1 == svec2 (d) "svec1[0] == "stone"

练习 14.4: 如何确定下列运算符是否应该是类的成员？

- (a) % (b) %= (c) ++ (d) -> (e) << (f) && (g) == (h) ()

练习 14.5: 在 7.5.1 节的练习 7.40 (第 261 页) 中，编写了下列类中某一个的框架，请问在这个类中应该定义重载的运算符吗？如果是，请写出来。

- (a) Book (b) Date (c) Employee
(d) Vehicle (e) Object (f) Tree



14.2 输入和输出运算符

如我们所知，IO 标准库分别使用 `>>` 和 `<<` 执行输入和输出操作。对于这两个运算符来说，IO 库定义了用其读写内置类型的版本，而类则需要自定义适合其对象的新版本以支持 IO 操作。



14.2.1 重载输出运算符 `<<`

通常情况下，输出运算符的第一个形参是一个非常量 `ostream` 对象的引用。之所以 `ostream` 是非常量是因为向流写入内容会改变其状态；而该形参是引用是因为我们无法直接复制一个 `ostream` 对象。

第二个形参一般来说是一个常量的引用，该常量是我们想要打印的类类型。第二个形参是引用的原因是我们希望避免复制实参；而之所以该形参可以是常量是因为（通常情况下）打印对象不会改变对象的内容。

为了与其他输出运算符保持一致，`operator<<` 一般要返回它的 `ostream` 形参。

`Sales_data` 的输出运算符

举个例子，我们按照如下形式编写 `Sales_data` 的输出运算符：

```
ostream &operator<<(ostream &os, const Sales_data &item)
{
    os << item.isbn() << " " << item.units_sold << " "
        << item.revenue << " " << item.avg_price();
    return os;
}
```

除了名字之外，这个函数与之前的 `print` 函数（参见 7.1.3 节，第 234 页）完全一样。打印一个 `Sales_data` 对象意味着要分别打印它的三个数据成员以及通过计算得到的平均销售价格，每个元素以空格隔开。完成输出后，运算符返回刚刚使用的 `ostream` 的引用。

输出运算符尽量减少格式化操作

用于内置类型的输出运算符不太考虑格式化操作，尤其不会打印换行符，用户希望类的输出运算符也像如此行事。如果运算符打印了换行符，则用户就无法在对象的同一行内接着打印一些描述性的文本了。相反，令输出运算符尽量减少格式化操作可以使用户有权控制输出的细节。

Best Practices

通常，输出运算符应该主要负责打印对象的内容而非控制格式，输出运算符不应该打印换行符。

输入输出运算符必须是非成员函数

与 `iostream` 标准库兼容的输入输出运算符必须是普通的非成员函数，而不能是类的成员函数。否则，它们的左侧运算对象将是我们的类的一个对象：

```
Sales_data data;
data << cout; // 如果 operator<< 是 Sales_data 的成员
```

假设输入输出运算符是某个类的成员，则它们也必须是 `istream` 或 `ostream` 的成员。然而，这两个类属于标准库，并且我们无法给标准库中的类添加任何成员。

因此, 如果我们希望为类自定义 IO 运算符, 则必须将其定义成非成员函数。当然, IO 运算符通常需要读写类的非公有数据成员, 所以 IO 运算符一般被声明为友元(参见 7.2.1 节, 第 241 页)。

558

14.2.1 节练习

练习 14.6: 为你的 Sales_data 类定义输出运算符。

练习 14.7: 你在 13.5 节的练习(第 470 页)中曾经编写了一个 String 类, 为它定义一个输出运算符。

练习 14.8: 你在 7.5.1 节的练习 7.40(第 261 页)中曾经选择并编写了一个类, 为它定义一个输出运算符。

14.2.2 重载输入运算符>>



通常情况下, 输入运算符的第一个形参是运算符将要读取的流的引用, 第二个形参是正要读入到的(非常量)对象的引用。该运算符通常会返回某个给定流的引用。第二个形参之所以必须是个非常量是因为输入运算符本身的目的就是将数据读入到这个对象中。

Sales_data 的输入运算符

举个例子, 我们将按照如下形式编写 Sales_data 的输入运算符:

```
istream &operator>>(istream &is, Sales_data &item)
{
    double price; // 不需要初始化, 因为我们将先读入数据到 price, 之后才使用它
    is >> item.bookNo >> item.units_sold >> price;
    if (is) // 检查输入是否成功
        item.revenue = item.units_sold * price;
    else
        item = Sales_data(); // 输入失败: 对象被赋予默认的状态
    return is;
}
```

除了 if 语句之外, 这个定义与之前的 read 函数(参见 7.1.3 节, 第 234 页)完全一样。if 语句检查读取操作是否成功, 如果发生了 IO 错误, 则运算符将给定的对象重置为空 Sales_data, 这样可以确保对象处于正确的状态。

Note

输入运算符必须处理输入可能失败的情况, 而输出运算符不需要。

输入时的错误

559

在执行输入运算符时可能发生下列错误:

- 当流含有错误类型的数据时读取操作可能失败。例如在读取完 bookNo 后, 输入运算符假定接下来读入的是两个数字数据, 一旦输入的不是数字数据, 则读取操作及后续对流的其他使用都将失败。
- 当读取操作到达文件末尾或者遇到输入流的其他错误时也会失败。

在程序中我们没有逐个检查每个读取操作, 而是等读取了所有数据后赶在使用这些数据前一次性检查:

```

if (is)                                // 检查输入是否成功
    item.revenue = item.units_sold * price;
else
    item = Sales_data();                // 输入失败: 对象被赋予默认的状态

```

如果读取操作失败, 则 `price` 的值将是未定义的。因此, 在使用 `price` 前我们需要首先检查输入流的合法性, 然后才能执行计算并将结果存入 `revenue`。如果发生了错误, 我们无须在意到底是哪部分输入失败, 只要将一个新的默认初始化的 `Sales_data` 对象赋予 `item` 从而将其重置为空 `Sales_data` 就可以了。执行这样的赋值后, `item` 的 `bookNo` 成员将是一个空 `string`, `revenue` 和 `units_sold` 成员将等于 0。

如果在发生错误前对象已经有一部分被改变, 则适时地将对象置为合法状态显得异常重要。例如在这个输入运算符中, 我们可能在成功读取新的 `bookNo` 后遇到错误, 这意味着对象的 `units_sold` 和 `revenue` 成员并没有改变, 因此有可能会将这两个数据与一条完全不匹配的 `bookNo` 组合在一起。

通过将对象置为合法的状态, 我们能 (略微) 保护使用者免于受到输入错误的影响。此时的对象处于可用状态, 即它的成员都是被正确定义的。而且该对象也不会产生误导性的结果, 因为它的数据在本质上确实是一体的。

Best
Practices

当读取操作发生错误时, 输入运算符应该负责从错误中恢复。

标示错误

一些输入运算符需要做更多数据验证的工作。例如, 我们的输入运算符可能需要检查 `bookNo` 是否符合规范的格式。在这样的例子中, 即使从技术上来看 IO 是成功的, 输入运算符也应该设置流的条件状态以标示出失败信息 (参见 8.1.2 节, 第 279 页)。通常情况下, 输入运算符只设置 `failbit`。除此之外, 设置 `eofbit` 表示文件耗尽, 而设置 `badbit` 表示流被破坏。最好的方式是由 IO 标准库自己来标示这些错误。

560

14.2.2 节练习

练习 14.9: 为你的 `Sales_data` 类定义输入运算符。

练习 14.10: 对于 `Sales_data` 的输入运算符来说如果给定了下面的输入将发生什么情况?

(a) 0-201-99999-9 10 24.95 (b) 10 24.95 0-210-99999-9

练习 14.11: 下面的 `Sales_data` 输入运算符存在错误吗? 如果有, 请指出来。对于这个输入运算符如果仍然给定上个练习的输入将发生什么情况?

```

istream& operator>>(istream& in, Sales_data& s)
{
    double price;
    in >> s.bookNo >> s.units_sold >> price;
    s.revenue = s.units_sold * price;
    return in;
}

```

练习 14.12: 你在 7.5.1 节的练习 7.40 (第 261 页) 中曾经选择并编写了一个类, 为它定义一个输入运算符并确保该运算符可以处理输入错误。

14.3 算术和关系运算符

通常情况下，我们把算术和关系运算符定义成非成员函数以允许对左侧或右侧的运算对象进行转换（参见 14.1 节，第 492 页）。因为这些运算符一般不需要改变运算对象的状态，所以形参都是常量的引用。

算术运算符通常会计算它的两个运算对象并得到一个新值，这个值有别于任意一个运算对象，常常位于一个局部变量之内，操作完成后返回该局部变量的副本作为其结果。如果类定义了算术运算符，则它一般也会定义一个对应的复合赋值运算符。此时，最有效的方式是使用复合赋值来定义算术运算符：

```
// 假设两个对象指向同一本书
Sales_data
operator+(const Sales_data &lsh, const Sales_data &rsh)
{
    Sales_data sum = lsh;           // 把 lsh 的数据成员拷贝给 sum
    sum += rsh;                     // 将 rsh 加到 sum 中
    return sum;
}
```

这个定义与原来的 `add` 函数（参见 7.1.3 节，第 234 页）是完全等价的。我们把 `lsh` 拷贝给局部变量 `sum`，然后使用 `Sales_data` 的复合赋值运算符（将在第 500 页定义）将 `rsh` 的值加到 `sum` 中，最后函数返回 `sum` 的副本。



如果类同时定义了算术运算符和相关的复合赋值运算符，则通常情况下应该使用复合赋值来实现算术运算符。

561

14.3 节练习

练习 14.13: 你认为 `Sales_data` 类还应该支持哪些其他算术运算符（参见表 4.1，第 124 页）？如果有的话，请给出它们的定义。

练习 14.14: 你觉得为什么调用 `operator+=` 来定义 `operator+` 比其他方法更有效？

练习 14.15: 你在 7.5.1 节的练习 7.40（第 261 页）中曾经选择并编写了一个类，你认为它应该含有其他算术运算符吗？如果是，请实现它们；如果不是，解释原因。

14.3.1 相等运算符

通常情况下，C++ 中的类通过定义相等运算符来检验两个对象是否相等。也就是说，它们会比较对象的每一个数据成员，只有当所有对应的成员都相等时才认为两个对象相等。依据这一思想，我们的 `Sales_data` 类的相等运算符不但应该比较 `bookNo`，还应该比较具体的销售数据：

```
bool operator==(const Sales_data &lsh, const Sales_data &rsh)
{
    return lsh.isbn() == rsh.isbn() &&
           lsh.units_sold == rsh.units_sold &&
           lsh.revenue == rsh.revenue;
}

bool operator!=(const Sales_data &lsh, const Sales_data &rsh)
```

```
{
    return !(lhs == rhs);
}
```

就上面这些函数的定义本身而言，它们似乎比较简单，也没什么价值，对于我们来说重要的是从这些函数中体现出来的设计准则：

- 如果一个类含有判断两个对象是否相等的操作，则它显然应该把函数定义成 `operator==` 而非一个普通的命名函数：因为用户肯定希望能使用 `==` 比较对象，所以提供了 `==` 就意味着用户无须再费时费力地学习并记忆一个全新的函数名字。此外，类定义了 `==` 运算符之后也更容易使用标准库容器和算法。
- 如果类定义了 `operator==`，则该运算符应该能判断一组给定的对象中是否含有重复数据。
- 通常情况下，相等运算符应该具有传递性，换句话说，如果 `a==b` 和 `b==c` 都为真，则 `a==c` 也应该为真。
- 如果类定义了 `operator==`，则这个类也应该定义 `operator!=`。对于用户来说，当他们能使用 `==` 时肯定也希望使用 `!=`，反之亦然。
- 相等运算符和不相等运算符中的一个应该把工作委托给另外一个，这意味着其中一个运算符应该负责实际比较对象的工作，而另一个运算符则只是调用那个真正工作的运算符。

562

Best
Practices

如果某个类在逻辑上有相等性的含义，则该类应该定义 `operator==`，这样可以使得用户更容易使用标准库算法来处理这个类。

14.3.1 节练习

练习 14.16: 为你的 `StrBlob` 类（参见 12.1.1 节，第 405 页）、`StrBlobPtr` 类（参见 12.1.6 节，第 421 页）、`StrVec` 类（参见 13.5 节，第 465 页）和 `String` 类（参见 13.5 节，第 470 页）分别定义相等运算符和不相等运算符。

练习 14.17: 你在 7.5.1 节的练习 7.40（第 261 页）中曾经选择并编写了一个类，你认为它应该含有相等运算符吗？如果是，请实现它；如果不是，解释原因。



14.3.2 关系运算符

定义了相等运算符的类也常常（但不总是）包含关系运算符。特别是，因为关联容器和一些算法要用到小于运算符，所以定义 `operator<` 会比较有用。

通常情况下关系运算符应该

1. 定义顺序关系，令其与关联容器中对关键字的要求一致（参见 11.2.2 节，第 378 页）；并且
2. 如果类同时也含有 `==` 运算符的话，则定义一种关系令其与 `==` 保持一致。特别是，如果两个对象是 `!=` 的，那么一个对象应该 `<` 另外一个。



尽管我们可能会认为 `Sales_data` 类应该支持关系运算符，但事实证明并非如此，其中的缘由比较微妙，值得读者深思。

一开始我们可能会认为应该像 `compareIsbn`（参见 11.2.2 节，第 379 页）那样定义 `<`，该函数通过比较 ISBN 来实现对两个对象的比较。然而，尽管 `compareIsbn` 提供的

顺序关系符合要求 1, 但是函数得到的结果显然与我们定义的 `==` 不一致, 因此它不满足要求 2。

对于 `Sales_data` 的 `==` 运算符来说, 如果两笔交易的 `revenue` 和 `units_sold` 成员不同, 那么即使它们的 `ISBN` 相同也无济于事, 它们仍然是不相等的。如果我们定义的 `<` 运算符仅仅比较 `ISBN` 成员, 那么将发生这样的情况: 两个 `ISBN` 相同但 `revenue` 和 `units_sold` 不同的对象经比较是不相等的, 但是其中的任何一个都不比另一个小。然而实际情况是, 如果我们有两个对象并且哪个都不比另一个小, 则从道理上来讲这两个对象应该是相等的。

< 563

基于上述分析我们也许会认为, 只要让 `operator<` 依次比较每个数据元素就能解决问题了, 比方说让 `operator<` 先比较 `isbn`, 相等的话继续比较 `units_sold`, 还相等再继续比较 `revenue`。

然而, 这样的排序没有任何必要。根据将来使用 `Sales_data` 类的实际需要, 我们可能会希望先比较 `units_sold`, 也可能希望先比较 `revenue`。有的时候, 我们希望 `units_sold` 少的对象“小于”`units_sold` 多的对象; 另一些时候, 则可能希望 `revenue` 少的对象“小于”`revenue` 多的对象。

因此对于 `Sales_data` 类来说, 不存在一种逻辑可靠的 `<` 定义, 这个类不定义 `<` 运算符也许更好。

Best
Practices

如果存在唯一一种逻辑可靠的 `<` 定义, 则应该考虑为这个类定义 `<` 运算符。如果类同时还包含 `==`, 则当且仅当 `<` 的定义和 `==` 产生的结果一致时才定义 `<` 运算符。

14.3.2 节练习

练习 14.18: 为你的 `StrBlob` 类、`StrBlobPtr` 类、`StrVec` 类和 `String` 类定义关系运算符。

练习 14.19: 你在 7.5.1 节的练习 7.40 (第 261 页) 中曾经选择并编写了一个类, 你认为它应该含有关系运算符吗? 如果是, 请实现它; 如果不是, 解释原因。

14.4 赋值运算符

之前已经介绍过拷贝赋值和移动赋值运算符 (参见 13.1.2 节, 第 443 页和 13.6.2 节, 第 474 页), 它们可以把类的一个对象赋值给该类的另一个对象。此外, 类还可以定义其他赋值运算符以使用别的类型作为右侧运算对象。

举个例子, 在拷贝赋值和移动赋值运算符之外, 标准库 `vector` 类还定义了第三种赋值运算符, 该运算符接受花括号内的元素列表作为参数 (参见 9.2.5 节, 第 302 页)。我们能以如下的形式使用该运算符:

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

同样, 也可以把这个运算符添加到 `StrVec` 类中 (参见 13.5 节, 第 465 页):

```
class StrVec {  
public:  
    StrVec &operator=(std::initializer_list<std::string>);  
    // 其他成员与 13.5 节 (第 465 页) 一致  
};
```

564 为了与内置类型的赋值运算符保持一致（也与我们已经定义的拷贝赋值和移动赋值运算一致），这个新的赋值运算符将返回其左侧运算对象的引用：

```
StrVec &StrVec::operator=(initializer_list<string> il)
{
    // alloc_n_copy 分配内存空间并从给定范围内拷贝元素
    auto data = alloc_n_copy(il.begin(), il.end());
    free(); // 销毁对象中的元素并释放内存空间
    elements = data.first; // 更新数据成员使其指向新空间
    first_free = cap = data.second;
    return *this;
}
```

和拷贝赋值及移动赋值运算符一样，其他重载的赋值运算符也必须先释放当前内存空间，再创建一片新空间。不同之处是，这个运算符无须检查对象向自身的赋值，这是因为它的形参 `initializer_list<string>`（参见 6.2.6 节，第 198 页）确保 `il` 与 `this` 所指的不是同一个对象。



我们可以重载赋值运算符。不论形参的类型是什么，赋值运算符都必须定义为成员函数。

复合赋值运算符

复合赋值运算符非得是类的成员，不过我们还是倾向于把包括复合赋值在内的所有赋值运算都定义在类的内部。为了与内置类型的复合赋值保持一致，类中的复合赋值运算符也要返回其左侧运算对象的引用。例如，下面是 `Sales_data` 类中复合赋值运算符的定义：

```
// 作为成员的二元运算符：左侧运算对象绑定到隐式的 this 指针
// 假定两个对象表示的是同一本书
Sales_data& Sales_data::operator+=(const Sales_data &rhs)
{
    units_sold += rhs.units_sold;
    revenue += rhs.revenue;
    return *this;
}
```



赋值运算符必须定义成类的成员，复合赋值运算符通常情况下也应该这样做。这两类运算符都应该返回左侧运算对象的引用。

14.4 节练习

练习 14.20：为你的 `Sales_data` 类定义加法和复合赋值运算符。

练习 14.21：编写 `Sales_data` 类的 `+` 和 `+=` 运算符，使得 `+` 执行实际的加法操作而 `+=` 调用 `+`。相比于 14.3 节（第 497 页）和 14.4 节（第 500 页）对这两个运算符的定义，本题的定义有何缺点？试讨论之。

练习 14.22：定义赋值运算符的一个新版本，使得我们能把一个表示 ISBN 的 `string` 赋给一个 `Sales_data` 对象。

练习 14.23：为你的 `StrVec` 类定义一个 `initializer_list` 赋值运算符。

练习 14.24: 你在 7.5.1 节的练习 7.40 (第 261 页) 中曾经选择并编写了一个类, 你认为它应该含有拷贝赋值和移动赋值运算符吗? 如果是, 请实现它们。

练习 14.25: 上题的这个类还需要定义其他赋值运算符吗? 如果是, 请实现它们; 同时说明运算对象应该是什么类型并解释原因。

14.5 下标运算符

表示容器的类通常可以通过元素在容器中的位置访问元素, 这些类一般会定义下标运算符 `operator[]`。



下标运算符必须是成员函数。

565

为了与下标的原始定义兼容, 下标运算符通常以所访问元素的引用作为返回值, 这样做的好处是下标可以出现在赋值运算符的任意一端。进一步, 我们最好同时定义下标运算符的常量版本和非常量版本, 当作用于一个常量对象时, 下标运算符返回常量引用以确保我们不会给返回的对象赋值。

Best Practices

如果一个类包含下标运算符, 则它通常会定义两个版本: 一个返回普通引用, 另一个是类的常量成员并且返回常量引用。

举个例子, 我们按照如下形式定义 `StrVec` (参见 13.5 节, 第 465 页) 的下标运算符:

```
class StrVec {
public:
    std::string& operator[](std::size_t n)
    { return elements[n]; }
    const std::string& operator[](std::size_t n) const
    { return elements[n]; }
    // 其他成员与 13.5 (第 465 页) 一致
private:
    std::string *elements; // 指向数组首元素的指针
};
```

上面这两个下标运算符的用法类似于 `vector` 或者数组中的下标。因为下标运算符返回的是元素的引用, 所以当 `StrVec` 是非常量时, 我们可以给元素赋值; 而当我们对常量对象取下标时, 不能为其赋值:

```
// 假设 svec 是一个 StrVec 对象
const StrVec cvec = svec; // 把 svec 的元素拷贝到 cvec 中
// 如果 svec 中含有元素, 对第一个元素运行 string 的 empty 函数
if (svec.size() && svec[0].empty()) {
    svec[0] = "zero"; // 正确: 下标运算符返回 string 的引用
    cvec[0] = "Zip"; // 错误: 对 cvec 取下标返回的是常量引用
}
```

566

14.5 节练习

练习 14.26: 为你的 StrBlob 类、StrBlobPtr 类、StrVec 类和 String 类定义下标运算符。

14.6 递增和递减运算符

在迭代器类中通常会实现递增运算符(++)和递减运算符(--),这两种运算符使得类可以在元素的序列中前后移动。C++语言并不要求递增和递减运算符必须是类的成员,但是因为它们改变的正好是所操作对象的状态,所以建议将其设定为成员函数。

对于内置类型来说,递增和递减运算符既有前置版本也有后置版本。同样,我们也应该为类定义两个版本的递增和递减运算符。接下来我们首先介绍前置版本,然后实现后置版本。

Best Practices

定义递增和递减运算符的类应该同时定义前置版本和后置版本。这些运算符通常应该被定义成类的成员。

定义前置递增/递减运算符

为了说明递增和递减运算符,我们不妨在 StrBlobPtr 类(参见 12.1.6 节,第 421 页)中定义它们:

```
class StrBlobPtr {
public:
    // 递增和递减运算符
    StrBlobPtr& operator++();           // 前置运算符
    StrBlobPtr& operator--();
    // 其他成员和之前的版本一致
};
```

Best Practices

为了与内置版本保持一致,前置运算符应该返回递增或递减后对象的引用。

567

递增和递减运算符的工作机理非常相似:它们首先调用 check 函数检验 StrBlobPtr 是否有效,如果是,接着检查给定的索引值是否有效。如果 check 函数没有抛出异常,则运算符返回对象的引用。

在递增运算符的例子中,我们把 curr 的当前值传递给 check 函数。如果这个值小于 vector 的大小,则 check 正常返回;否则,如果 curr 已经到达了 vector 的末尾,check 将抛出异常:

```
// 前置版本:返回递增/递减对象的引用
StrBlobPtr& StrBlobPtr::operator++()
{
    // 如果 curr 已经指向了容器的尾后位置,则无法递增它
    check(curr, "increment past end of StrBlobPtr");
    ++curr;           // 将 curr 在当前状态下向前移动一个元素
    return *this;
}
```

```
StrBlobPtr& StrBlobPtr::operator--()
{
    // 如果 curr 是 0, 则继续递减它将产生一个无效下标
    --curr; // 将 curr 在当前状态下向后移动一个元素
    check(curr, "decrement past begin of StrBlobPtr");
    return *this;
}
```

递减运算符先递减 `curr`, 然后调用 `check` 函数。此时, 如果 `curr` (一个无符号数) 已经是 0 了, 那么我们传递给 `check` 的值将是一个表示无效下标的非常大的正数值 (参见 2.1.2 节, 第 33 页)。

区分前置和后置运算符

要想同时定义前置和后置运算符, 必须首先解决一个问题, 即普通的重载形式无法区分这两种情况。前置和后置版本使用的是同一个符号, 意味着其重载版本所用的名字将是相同的, 并且运算对象的数量和类型也相同。

为了解决这个问题, 后置版本接受一个额外的 (不被使用) `int` 类型的形参。当我们使用后置运算符时, 编译器为这个形参提供一个值为 0 的实参。尽管从语法上来说后置函数可以使用这个额外的形参, 但是在实际过程中通常不会这么做。这个形参的唯一作用就是区分前置版本和后置版本的函数, 而不是真的要在实现后置版本时参与运算。

接下来我们为 `StrBlobPtr` 添加后置运算符:

```
class StrBlobPtr {
public:
    // 递增和递减运算符
    StrBlobPtr operator++(int); // 后置运算符
    StrBlobPtr operator--(int);
    // 其他成员和之前的版本一致
};
```

Best
Practices

为了与内置版本保持一致, 后置运算符应该返回对象的原值 (递增或递减之前的值), 返回的形式是一个值而非引用。

568

对于后置版本来说, 在递增对象之前需要首先记录对象的状态:

```
// 后置版本: 递增/递减对象的值但是返回原值
StrBlobPtr StrBlobPtr::operator++(int)
{
    // 此处无须检查有效性, 调用前置递增运算时才需要检查
    StrBlobPtr ret = *this; // 记录当前的值
    ++*this; // 向前移动一个元素, 前置++需要检查递增的有效性
    return ret; // 返回之前记录的状态
}

StrBlobPtr StrBlobPtr::operator--(int)
{
    // 此处无须检查有效性, 调用前置递减运算时才需要检查
    StrBlobPtr ret = *this; // 记录当前的值
    --*this; // 向后移动一个元素, 前置--需要检查递减的有效性
    return ret; // 返回之前记录的状态
}
```

由上可知，我们的后置运算符调用各自的前置版本来完成实际的工作。例如后置递增运算符执行

```
++*this
```

该表达式调用前置递增运算符，前置递增运算符首先检查递增操作是否安全，根据检查的结果抛出一个异常或者执行递增 `curr` 的操作。假定通过了检查，则后置函数返回事先存好的 `ret` 的副本。因此最终的效果是，对象本身向前移动了一个元素，而返回的结果仍然反映对象在未递增之前原始的值。

Note

因为我们不会用到 `int` 形参，所以无须为其命名。

显式地调用后置运算符

如在第 491 页介绍的，可以显式地调用一个重载的运算符，其效果与在表达式中以运算符的形式使用它完全一样。如果我们想通过函数调用的方式调用后置版本，则必须为它的整型参数传递一个值：

```
StrBlobPtr p(a1);           // p 指向 a1 中的 vector
p.operator++(0);             // 调用后置版本的 operator++
p.operator++();              // 调用前置版本的 operator++
```

尽管传入的值通常会被运算符函数忽略，但却必不可少，因为编译器只有通过它才能知道应该使用后置版本。

569

14.6 节练习

练习 14.27: 为你的 `StrBlobPtr` 类添加递增和递减运算符。

练习 14.28: 为你的 `StrBlobPtr` 类添加加法和减法运算符，使其可以实现指针的算术运算（参见 3.5.3 节，第 106 页）。

练习 14.29: 为什么不定义 `const` 版本的递增和递减运算符？

14.7 成员访问运算符

在迭代器类及智能指针类（参见 12.1 节，第 400 页）中常常用到解引用运算符（`*`）和箭头运算符（`->`）。我们以如下形式向 `StrBlobPtr` 类添加这两种运算符：

```
class StrBlobPtr {
public:
    std::string& operator*() const
    { auto p = check(curr, "dereference past end");
      return (*p)[curr];           // (*p) 是对象所指的 vector
    }

    std::string* operator->() const
    { // 将实际工作委托给解引用运算符
      return & this->operator*();
    }

    // 其他成员与之前的版本一致
}
```

解引用运算符首先检查 `curr` 是否仍在作用范围内, 如果是, 则返回 `curr` 所指元素的一个引用。箭头运算符不执行任何自己的操作, 而是调用解引用运算符并返回解引用结果元素的地址。



箭头运算符必须是类的成员。解引用运算符通常也是类的成员, 尽管并非必须如此。

值得注意的是, 我们将这两个运算符定义成了 `const` 成员, 这是因为与递增和递减运算符不一样, 获取一个元素并不会改变 `StrBlobPtr` 对象的状态。同时, 它们的返回值分别是非常量 `string` 的引用或指针, 因为一个 `StrBlobPtr` 只能绑定到非常量的 `StrBlob` 对象 (参见 12.1.6 节, 第 421 页)。

这两个运算符的用法与指针或者 `vector` 迭代器的对应操作完全一致:

```
StrBlob a1 = {"hi", "bye", "now"};
StrBlobPtr p(a1);           // p 指向 a1 中的 vector
*p = "okay";                 // 给 a1 的首元素赋值
cout << p->size() << endl;    // 打印 4, 这是 a1 首元素的大小
cout << (*p).size() << endl;  // 等价于 p->size()
```

对箭头运算符返回值的限定

570

和大多数其他运算符一样 (尽管这么做不太好), 我们能令 `operator*` 完成任何我们指定的操作。换句话说, 我们可以让 `operator*` 返回一个固定值 42, 或者打印对象的内容, 或者其他。箭头运算符则不是这样, 它永远不能丢掉成员访问这个最基本的含义。当我们重载箭头时, 可以改变的是箭头从哪个对象当中获取成员, 而箭头获取成员这一事实则永远不变。

对于形如 `point->mem` 的表达式来说, `point` 必须是指向类对象的指针或者是一个重载了 `operator->` 的类的对象。根据 `point` 类型的不同, `point->mem` 分别等价于

```
(*point).mem;           // point 是一个内置的指针类型
point.operator()->mem;    // point 是类的一个对象
```

除此之外, 代码都将发生错误。 `point->mem` 的执行过程如下所示:

1. 如果 `point` 是指针, 则我们应用内置的箭头运算符, 表达式等价于 `(*point).mem`。首先解引用该指针, 然后从所得的对象中获取指定的成员。如果 `point` 所指的类型没有名为 `mem` 的成员, 程序会发生错误。
2. 如果 `point` 是定义了 `operator->` 的类的一个对象, 则我们使用 `point.operator->()` 的结果来获取 `mem`。其中, 如果该结果是一个指针, 则执行第 1 步; 如果该结果本身含有重载的 `operator->()`, 则重复调用当前步骤。最终, 当这一过程结束时程序或者返回了所需的内容, 或者返回一些表示程序错误的信息。



重载的箭头运算符必须返回类的指针或者自定义了箭头运算符的某个类的对象。

14.7 节练习

练习 14.30: 为你的 `StrBlobPtr` 类和在 12.1.6 节练习 12.22 (第 423 页) 中定义的 `ConstStrBlobPtr` 类分别添加解引用运算符和箭头运算符。注意: 因为 `ConstStrBlobPtr` 的数据成员指向 `const vector`, 所以 `ConstStrBlobPtr` 中的运算符必须返回常量引用。

练习 14.31: 我们的 `StrBlobPtr` 类没有定义拷贝构造函数、赋值运算符及析构函数, 为什么?

练习 14.32: 定义一个类令其含有指向 `StrBlobPtr` 对象的指针, 为这个类定义重载的箭头运算符。



14.8 函数调用运算符

571

如果类重载了函数调用运算符, 则我们可以像使用函数一样使用该类的对象。因为这样的类同时也能存储状态, 所以与普通函数相比它们更加灵活。

举个简单的例子, 下面这个名为 `absInt` 的 `struct` 含有一个调用运算符, 该运算符负责返回其参数的绝对值:

```
struct absInt {
    int operator()(int val) const {
        return val < 0 ? -val : val;
    }
};
```

这个类只定义了一种操作: 函数调用运算符, 它负责接受一个 `int` 类型的实参, 然后返回该实参的绝对值。

我们使用调用运算符的方式是令一个 `absInt` 对象作用于一个实参列表, 这一过程看起来非常像调用函数的过程:

```
int i = -42;
absInt absObj;           // 含有函数调用运算符的对象
int ui = absObj(i);       // 将 i 传递给 absObj.operator()
```

即使 `absObj` 只是一个对象而非函数, 我们也能“调用”该对象。调用对象实际上是在运行重载的调用运算符。在此例中, 该运算符接受一个 `int` 值并返回其绝对值。



函数调用运算符必须是成员函数。一个类可以定义多个不同版本的调用运算符, 相互之间应该在参数数量或类型上有所区别。

如果类定义了调用运算符, 则该类的对象称作函数对象 (function object)。因为可以调用这种对象, 所以我们说这些对象的“行为像函数一样”。

含有状态的函数对象类

和其他类一样, 函数对象类除了 `operator()` 之外也可以包含其他成员。函数对象类通常含有一些数据成员, 这些成员被用于定制调用运算符中的操作。

举个例子, 我们将定义一个打印 `string` 实参内容的类。默认情况下, 我们的类会将

内容写入到 `cout` 中，每个 `string` 之间以空格隔开。同时也允许类的用户提供其他可写入的流及其他分隔符。我们将该类定义如下：

```
class PrintString {
public:
    PrintString(ostream &o = cout, char c = ' '):
        os(o), sep(c) { }
    void operator()(const string &s) const { os << s << sep; }
private:
    ostream &os;           // 用于写入的目的流
    char sep;              // 用于将不同输出隔开的字符
};
```

我们的类有一个构造函数，它接受一个输出流的引用以及一个用于分隔的字符，这两个形参的默认实参（参见 6.5.1 节，第 211 页）分别是 `cout` 和空格。之后的函数调用运算符使用这些成员协助其打印给定的 `string`。

572

当定义 `PrintString` 的对象时，对于分隔符及输出流既可以使用默认值也可以提供我们自己的值：

```
PrintString printer;           // 使用默认值，打印到 cout
printer(s);                   // 在 cout 中打印 s，后面跟一个空格
PrintString errors(cerr, '\n');
errors(s);                     // 在 cerr 中打印 s，后面跟一个换行符
```

函数对象常常作为泛型算法的实参。例如，可以使用标准库 `for_each` 算法（参见 10.3.2 节，第 348 页）和我们自己的 `PrintString` 类来打印容器的内容：

```
for_each(vs.begin(), vs.end(), PrintString(cerr, '\n'));
```

`for_each` 的第三个实参是类型 `PrintString` 的一个临时对象，其中我们用 `cerr` 和换行符初始化了该对象。当程序调用 `for_each` 时，将会把 `vs` 中的每个元素依次打印到 `cerr` 中，元素之间以换行符分隔。

14.8 节练习

练习 14.33：一个重载的函数调用运算符应该接受几个运算对象？

练习 14.34：定义一个函数对象类，令其执行 `if-then-else` 的操作：该类的调用运算符接受三个形参，它首先检查第一个形参，如果成功返回第二个形参的值；如果不成功返回第三个形参的值。

练习 14.35：编写一个类似于 `PrintString` 的类，令其从 `istream` 中读取一行输入，然后返回一个表示我们所读内容的 `string`。如果读取失败，返回空 `string`。

练习 14.36：使用前一个练习定义的类读取标准输入，将每一行保存为 `vector` 的一个元素。

练习 14.37：编写一个类令其检查两个值是否相等。使用该对象及标准库算法编写程序，令其替换某个序列中具有给定值的所有实例。

14.8.1 `lambda` 是函数对象

在前一节中，我们使用一个 `PrintString` 对象作为调用 `for_each` 的实参，这一

用法类似于我们在 10.3.2 节（第 346 页）中编写的使用 `lambda` 表达式的程序。当我们编写了一个 `lambda` 后，编译器将该表达式翻译成一个未命名类的未命名对象（参见 10.3.3 节，第 349 页）。在 `lambda` 表达式产生的类中含有一个重载的函数调用运算符，例如，对于我们传递给 `stable_sort` 作为其最后一个实参的 `lambda` 表达式来说：

```
// 根据单词的长度对其进行排序，对于长度相同的单词按照字母表顺序排序
stable_sort(words.begin(), words.end(),
            [](const string &a, const string &b)
            { return a.size() < b.size(); });
```

其行为类似于下面这个类的一个未命名对象

```
class ShorterString {
public:
    bool operator()(const string &s1, const string &s2) const
    { return s1.size() < s2.size(); }
};
```

产生的类只有一个函数调用运算符成员，它负责接受两个 `string` 并比较它们的长度，它的形参列表和函数体与 `lambda` 表达式完全一样。如我们在 10.3.3 节（第 352 页）所见，默认情况下 `lambda` 不能改变它捕获的变量。因此在默认情况下，由 `lambda` 产生的类当中的函数调用运算符是一个 `const` 成员函数。如果 `lambda` 被声明为可变的，则调用运算符就不是 `const` 的了。

用这个类替代 `lambda` 表达式后，我们可以重写并重新调用 `stable_sort`：

```
stable_sort(words.begin(), words.end(), ShorterString());
```

第三个实参是新构建的 `ShorterString` 对象，当 `stable_sort` 内部的代码每次比较两个 `string` 时就会“调用”这一对象，此时该对象将调用运算符的函数体，判断第一个 `string` 的大小小于第二个时返回 `true`。

表示 `lambda` 及相应捕获行为的类

如我们所知，当一个 `lambda` 表达式通过引用捕获变量时，将由程序负责确保 `lambda` 执行时引用所引的对象确实存在（参见 10.3.3 节，第 350 页）。因此，编译器可以直接使用该引用而无须在 `lambda` 产生的类中将其存储为数据成员。

相反，通过值捕获的变量被拷贝到 `lambda` 中（参见 10.3.3 节，第 350 页）。因此，这种 `lambda` 产生的类必须为每个值捕获的变量建立对应的数据成员，同时创建构造函数，令其使用捕获的变量的值来初始化数据成员。举个例子，在 10.3.2 节（第 347 页）中有一个 `lambda`，它的作用是找到第一个长度不小于给定值的 `string` 对象：

```
// 获得第一个指向满足条件元素的迭代器，该元素满足 size() is >= sz
auto wc = find_if(words.begin(), words.end(),
                 [sz](const string &a)
                 { return a.size() >= sz; });
```

该 `lambda` 表达式产生的类将形如：

```
class SizeComp {
    SizeComp(size_t n): sz(n) { } // 该形参对应捕获的变量
    // 该调用运算符的返回类型、形参和函数体都与 lambda 一致
    bool operator()(const string &s) const
    { return s.size() >= sz; }
```

```
private:
    size_t sz; // 该数据成员对应通过值捕获的变量
};
```

和我们的 `ShorterString` 类不同，上面这个类含有一个数据成员以及一个用于初始化该成员的构造函数。这个合成的类不含有默认构造函数，因此要想使用这个类必须提供一个实参：

```
// 获得第一个指向满足条件元素的迭代器，该元素满足 size() is >= sz
auto wc = find_if(words.begin(), words.end(), SizeComp(sz));
```

`lambda` 表达式产生的类不含默认构造函数、赋值运算符及默认析构函数；它是否含有默认的拷贝/移动构造函数则通常要视捕获的数据成员类型而定（参见 13.1.6 节，第 450 页和 13.6.2 节，第 475 页）。

14.8.1 节练习

练习 14.38: 编写一个类令其检查某个给定的 `string` 对象的长度是否与一个阈值相等。使用该对象编写程序，统计并报告在输入的文件中长度为 1 的单词有多少个、长度为 2 的单词有多少个、……、长度为 10 的单词又有多少个。

练习 14.39: 修改上一题的程序令其报告长度在 1 至 9 之间的单词有多少个、长度在 10 以上的单词又有多少个。

练习 14.40: 重新编写 10.3.2 节（第 349 页）的 `biggies` 函数，使用函数对象类替换其中的 `lambda` 表达式。

练习 14.41: 你认为 C++11 新标准为什么要增加 `lambda`？对于你自己来说，什么情况下会使用 `lambda`，什么情况下会使用类？

14.8.2 标准库定义的函数对象

标准库定义了一组表示算术运算符、关系运算符和逻辑运算符的类，每个类分别定义了一个执行命名操作的调用运算符。例如，`plus` 类定义了一个函数调用运算符用于对一对象执行 `+` 的操作；`modulus` 类定义了一个调用运算符执行二元的 `%` 操作；`equal_to` 类执行 `==`，等等。

这些类都被定义成模板的形式，我们可以为其指定具体的应用类型，这里的类型即调用运算符的形参类型。例如，`plus<string>` 令 `string` 加法运算符作用于 `string` 对象；`plus<int>` 的运算对象是 `int`；`plus<Sales_data>` 对 `Sales_data` 对象执行加法运算，以此类推：

```
plus<int> intAdd; // 可执行 int 加法的函数对象
negate<int> intNegate; // 可对 int 值取反的函数对象
// 使用 intAdd::operator(int, int) 求 10 和 20 的和
int sum = intAdd(10, 20); // 等价于 sum = 30
sum = intNegate(intAdd(10, 20)); // 等价于 sum = 30
// 使用 intNegate::operator(int) 生成 -10
// 然后将 -10 作为 intAdd::operator(int, int) 的第二个参数
sum = intAdd(10, intNegate(10)); // sum = 0
```

表 14.2 所列的类型定义在 `functional` 头文件中。

表 14.2: 标准库函数对象		
算术	关系	逻辑
plus<Type>	equal_to<Type>	logical_and<Type>
minus<Type>	not_equal_to<Type>	logical_or<Type>
multiplies<Type>	greater<Type>	logical_not<Type>
divides<Type>	greater_equal<Type>	
modulus<Type>	less<Type>	
negate<Type>	less_equal<Type>	

在算法中使用标准库函数对象

表示运算符的函数对象类常用来替换算法中的默认运算符。如我们所知，在默认情况下排序算法使用 operator<将序列按照升序排列。如果要执行降序排列的话，我们可以传入一个 greater 类型的对象。该类将产生一个调用运算符并负责执行待排序类型的大于运算。例如，如果 svec 是一个 vector<string>，

```
// 传入一个临时的函数对象用于执行两个 string 对象的>比较运算
sort(svec.begin(), svec.end(), greater<string>());
```

则上面的语句将按照降序对 svec 进行排序。第三个实参是 greater<string>类型的一个未命名的对象，因此当 sort 比较元素时，不再是使用默认的<运算符，而是调用给定的 greater 函数对象。该对象负责在 string 元素之间执行>比较运算。

需要特别注意的是，标准库规定其函数对象对于指针同样适用。我们之前曾经介绍过比较两个无关指针将产生未定义的行为（参见 3.5.3 节，第 107 页），然而我们可能会希望通过比较指针的内存地址来 sort 指针的 vector。直接这么做将产生未定义的行为，因此我们可以使用一个标准库函数对象来实现该目的：

```
vector<string *> nameTable; // 指针的 vector
// 错误：nameTable 中的指针彼此之间没有关系，所以<将产生未定义的行为
sort(nameTable.begin(), nameTable.end(),
      [](string *a, string *b) { return a < b; });
// 正确：标准库规定指针的 less 是定义良好的
sort(nameTable.begin(), nameTable.end(), less<string*>());
```

576 关联容器使用 less<key_type>对元素排序，因此我们可以定义一个指针的 set 或者在 map 中使用指针作为关键值而无须直接声明 less。

14.8.2 节练习

练习 14.42: 使用标准库函数对象及适配器定义一条表达式，令其

(a) 统计大于 1024 的值有多少个。

(b) 找到第一个不等于 pooh 的字符串。

(c) 将所有的值乘以 2。

练习 14.43: 使用标准库函数对象判断一个给定的 int 值是否能被 int 容器中的所有元素整除。

14.8.3 可调用对象与 function

C++语言中有几种可调用的对象：函数、函数指针、lambda 表达式（参见 10.3.2 节，第 346 页）、bind 创建的对象（参见 10.3.4 节，第 354 页）以及重载了函数调用运算符的类。

和其他对象一样，可调用的对象也有类型。例如，每个 lambda 有它自己唯一的（未命名）类类型；函数及函数指针的类型则由其返回值类型和实参类型决定，等等。

然而，两个不同类型的可调用对象却可能共享同一种调用形式（call signature）。调用形式指明了调用返回的类型以及传递给调用的实参类型。一种调用形式对应一个函数类型，例如：

```
int(int, int)
```

是一个函数类型，它接受两个 int、返回一个 int。

不同类型可能具有相同的调用形式

对于几个可调用对象共享同一种调用形式的情况，有时我们会希望把它们看成具有相同的类型。例如，考虑下列不同类型的可调用对象：

```
// 普通函数
int add(int i, int j) { return i + j; }
// lambda，其产生一个未命名的函数对象类
auto mod = [](int i, int j) { return i % j; };
// 函数对象类
struct divide {
    int operator()(int denominator, int divisor) {
        return denominator / divisor;
    }
};
```

上面这些可调用对象分别对其参数执行了不同的算术运算，尽管它们的类型各不相同，但是共享同一种调用形式： ◀ 577

```
int(int, int)
```

我们可能希望使用这些可调用对象构建一个简单的桌面计算器。为了实现这一目的，需要定义一个函数表（function table）用于存储指向这些可调用对象的“指针”。当程序需要执行某个特定的操作时，从表中查找该调用的函数。

在 C++语言中，函数表很容易通过 map 来实现。对于此例来说，我们使用一个表示运算符符号的 string 对象作为关键字；使用实现运算符的函数作为值。当我们要求给定运算符的值时，先通过运算符索引 map，然后调用找到的那个元素。

假定我们的所有函数都相互独立，并且只处理关于 int 的二元运算，则 map 可以定义成如下的形式：

```
// 构建从运算符到函数指针的映射关系，其中函数接受两个 int、返回一个 int
map<string, int(*) (int,int)> binops;
```

我们可以按照下面的形式将 add 的指针添加到 binops 中：

```
// 正确：add 是一个指向正确类型函数的指针
binops.insert({"+", add}); // {"+", add} 是一个 pair（参见 11.2.3 节，379 页）
```

但是我们不能将 mod 或者 divide 存入 binops：

```
binops.insert({"%", mod}); // 错误: mod 不是一个函数指针
```

问题在于 `mod` 是个 `lambda` 表达式, 而每个 `lambda` 有它自己的类类型, 该类型与存储在 `binops` 中的值的类型不匹配。

标准库 `function` 类型

C++
11

我们可以使用一个名为 `function` 的新的标准库类型解决上述问题, `function` 定义在 `functional` 头文件中, 表 14.3 列举出了 `function` 定义的操作。

表 14.3: `function` 的操作

<code>function<T> f;</code>	<code>f</code> 是一个用来存储可调用对象的空 <code>function</code> , 这些可调用对象的调用形式应该与函数类型 <code>T</code> 相同 (即 <code>T</code> 是 <code>retType(args)</code>)
<code>function<T> f(nullptr);</code>	显式地构造一个空 <code>function</code>
<code>function<T> f(obj);</code>	在 <code>f</code> 中存储可调用对象 <code>obj</code> 的副本
<code>f</code>	将 <code>f</code> 作为条件: 当 <code>f</code> 含有一个可调用对象时为真; 否则为假
<code>f(args)</code>	调用 <code>f</code> 中的对象, 参数是 <code>args</code>

定义为 `function<T>` 的成员的类型

<code>result_type</code>	该 <code>function</code> 类型的可调用对象返回的类型
<code>argument_type</code>	当 <code>T</code> 有一个或两个实参时定义的类型。如果 <code>T</code> 只有一个实参, 则 <code>argument_type</code> 是该类型的同义词; 如果 <code>T</code> 有两个实参, 则 <code>first_argument_type</code> 和 <code>second_argument_type</code> 分别代表两个实参的类型
<code>first_argument_type</code>	
<code>second_argument_type</code>	

`function` 是一个模板, 和我们使用过的其他模板一样, 当创建一个具体的 `function` 类型时我们必须提供额外的信息。在此例中, 所谓额外的信息是指该 `function` 类型能够表示的对象的调用形式。参考其他模板, 我们在一对尖括号内指定类型:

```
function<int(int, int)>
```

在这里我们声明了一个 `function` 类型, 它可以表示接受两个 `int`、返回一个 `int` 的可调用对象。因此, 我们可以用这个新声明的类型表示任意一种桌面计算器用到的类型;

```
function<int(int, int)> f1 = add; // 函数指针
function<int(int, int)> f2 = divide(); // 函数对象类的对象
function<int(int, int)> f3 = [](int i, int j) // lambda
    { return i * j; };
cout << f1(4,2) << endl; // 打印 6
cout << f2(4,2) << endl; // 打印 2
cout << f3(4,2) << endl; // 打印 8
```

578

使用这个 `function` 类型我们可以重新定义 `map`:

```
// 列举了可调用对象与二元运算符对应关系的表格
// 所有可调用对象都必须接受两个 int、返回一个 int
// 其中的元素可以是函数指针、函数对象或者 lambda
map<string, function<int(int, int)>> binops;
```

我们能把所有可调用对象, 包括函数指针、`lambda` 或者函数对象在内, 都添加到这个 `map` 中:

```
map<string, function<int(int, int)>> binops = {
    {"+", add}, // 函数指针
    {"-", std::minus<int>()}, // 标准库函数对象
    {"/", divide()}, // 用户定义的函数对象
    {"*", [](int i, int j) { return i * j; }}, // 未命名的 lambda
    {"%", mod}; // 命名了的 lambda 对象
```

我们的 map 中包含 5 个元素，尽管其中的可调用对象的类型各不相同，我们仍然能够把所有这些类型都存储在同一个 `function<int (int , int)>` 类型中。

一如往常，当我们索引 map 时将得到关联值的一个引用。如果我们索引 binops，将得到 function 对象的引用。function 类型重载了调用运算符，该运算符接受它自己的实参然后将其传递给存好的可调用对象：

```
binops["+"](10, 5); // 调用 add(10, 5)
binops["-"](10, 5); // 使用 minus<int>对象的调用运算符
binops["/"](10, 5); // 使用 divide 对象的调用运算符
binops["*"](10, 5); // 调用 lambda 函数对象
binops["%"](10, 5); // 调用 lambda 函数对象
```

我们依次调用了 binops 中存储的每个操作。在第一个调用中，我们获得的元素存放着一个指向 add 函数的指针，因此调用 `binops["+"](10, 5)` 实际上是使用该指针调用 add，并传入 10 和 5。在接下来的调用中，`binops["-"]` 返回一个存放着 `std::minus<int>` 类型对象的 function，我们将执行该对象的调用运算符。

重载的函数与 function

我们不能（直接）将重载函数的名字存入 function 类型的对象中：

```
int add(int i, int j) { return i + j; }
Sales_data add(const Sales_data&, const Sales_data&);
map<string, function<int(int, int)>> binops;
binops.insert( {"+", add} ); // 错误：哪个 add?
```

解决上述二义性问题的一条途径是存储函数指针（参见 6.7 节，第 221 页）而非函数的名字：

```
int (*fp)(int,int) = add; // 指针所指的 add 是接受两个 int 的版本
binops.insert( {"+", fp} ); // 正确：fp 指向一个正确的 add 版本
```

同样，我们也能使用 lambda 来消除二义性：

```
// 正确：使用 lambda 来指定我们希望使用的 add 版本
binops.insert( {"+", [](int a, int b) {return add(a, b);}} );
```

lambda 内部的函数调用传入了两个 int，因此该调用只能匹配接受两个 int 的 add 版本，而这也正是执行 lambda 时真正调用的函数。



新版本标准库中的 function 类与旧版本中的 unary_function 和 binary_function 没有关联，后两个类已经被更通用的 bind 函数替代了（参见 10.3.4 节，第 357 页）。

14.8.3 节练习

练习 14.44: 编写一个简单的桌面计算器使其能处理二元运算。



14.9 重载、类型转换与运算符

在 7.5.4 节（第 263 页）中我们看到由一个实参调用的非显式构造函数定义了一种隐式的类型转换，这种构造函数将实参类型的对象转换成类类型。我们同样能定义对于类类型的类型转换，通过定义类型转换运算符可以做到这一点。转换构造函数和类型转换运算符共同定义了类类型转换（class-type conversions），这样的转换有时也被称作用户定义的类型转换（user-defined conversions）。

14.9.1 类型转换运算符

类型转换运算符（conversion operator）是类的一种特殊成员函数，它负责将一个类类型的值转换成其他类型。类型转换函数的一般形式如下所示：

```
operator type() const;
```

其中 *type* 表示某种类型。类型转换运算符可以面向任意类型（除了 `void` 之外）进行定义，只要该类型能作为函数的返回类型（参见 6.1 节，第 184 页）。因此，我们不允许转换成数组或者函数类型，但允许转换成指针（包括数组指针及函数指针）或者引用类型。

类型转换运算符既没有显式的返回类型，也没有形参，而且必须定义成类的成员函数。类型转换运算符通常不应该改变待转换对象的内容，因此，类型转换运算符一般被定义成 `const` 成员。



一个类型转换函数必须是类的成员函数；它不能声明返回类型，形参列表也必须为空。类型转换函数通常应该是 `const`。

定义含有类型转换运算符的类

举个例子，我们定义一个比较简单的类，令其表示 0 到 255 之间的一个整数：

```
class SmallInt {
public:
    SmallInt(int i = 0): val(i)
    {
        if (i < 0 || i > 255)
            throw std::out_of_range("Bad SmallInt value");
    }
    operator int() const { return val; }
private:
    std::size_t val;
};
```

我们的 `SmallInt` 类既定义了向类类型的转换，也定义了从类类型向其他类型的转换。其中，构造函数将算术类型的值转换成 `SmallInt` 对象，而类型转换运算符将 `SmallInt` 对象转换成 `int`：

```
SmallInt si;
```

```

si = 4;      // 首先将 4 隐式地转换成 SmallInt, 然后调用 SmallInt::operator=
si + 3;      // 首先将 si 隐式地转换成 int, 然后执行整数的加法

```

尽管编译器一次只能执行一个用户定义的类型转换（参见 4.11.2 节，第 144 页），但是隐式的用户定义类型转换可以置于一个标准（内置）类型转换之前或之后（参见 4.11.1 节，第 141 页），并与其一起使用。因此，我们可以将任何算术类型传递给 SmallInt 的构造函数。类似的，我们也能使用类型转换运算符将一个 SmallInt 对象转换成 int，然后再将所得的 int 转换成任何其他算术类型：

```

// 内置类型转换将 double 实参转换成 int
SmallInt si = 3.14;      // 调用 SmallInt(int) 构造函数
// SmallInt 的类型转换运算符将 si 转换成 int
si + 3.14;      // 内置类型转换将所得的 int 继续转换成 double

```

因为类型转换运算符是隐式执行的，所以无法给这些函数传递实参，当然也就不能在类型转换运算符的定义中使用任何形参。同时，尽管类型转换函数不负责指定返回类型，但实际上每个类型转换函数都会返回一个对应类型的值：

```

class SmallInt;
operator int(SmallInt&);      // 错误：不是成员函数
class SmallInt {
public:
    int operator int() const;      // 错误：指定了返回类型
    operator int(int = 0) const;    // 错误：参数列表不为空
    operator int*() const { return 42; } // 错误：42 不是一个指针
};

```

提示：避免过度使用类型转换函数

和使用重载运算符的经验一样，明智地使用类型转换运算符也能极大地简化类设计者的工作，同时使得使用类更加容易。然而，如果在类类型和转换类型之间不存在明显的映射关系，则这样的类型转换可能具有误导性。

例如，假设某个类表示 Date，我们也许会为它添加一个从 Date 到 int 的转换。然而，类型转换函数的返回值应该是什么？一种可能的解释是，函数返回一个十进制数，依次表示年、月、日，例如，July 30, 1989 可能转换为 int 值 19890730。同时还存在另外一种合理的解释，即类型转换运算符返回的 int 表示的是从某个时间节点（比如 January 1, 1970）开始经过的天数。显然这两种理解都合情合理，毕竟从形式上看它们产生的效果都是越靠后的日期对应的整数值越大，而且两种转换都有实际的用处。

问题在于 Date 类型的对象和 int 类型的值之间不存在明确的一对一映射关系。因此在此例中，不定义该类型转换运算符也许会更好。作为替代的手段，类可以定义一个或多个普通的成员函数以从各种不同形式中提取所需的信息。

类型转换运算符可能产生意外结果

在实践中，类很少提供类型转换运算符。在大多数情况下，如果类型转换自动发生，用户可能会感觉比较意外，而不是感觉受到了帮助。然而这条经验法则则存在一种例外情况：对于类来说，定义向 bool 的类型转换还是比较普遍的现象。

在 C++ 标准的早期版本中，如果类想定义一个向 bool 的类型转换，则它常常遇到一个问题：因为 bool 是一种算术类型，所以类类型的对象转换成 bool 后就能被用在任何

< 581

< 582

需要算术类型的上下文中。这样的类型转换可能引发意想不到的结果，特别是当 `istream` 含有向 `bool` 的类型转换时，下面的代码仍将编译通过：

```
int i = 42;
cin << i; // 如果向 bool 的类型转换不是显式的，则该代码在编译器看来将是合法的！
```

这段程序试图将输出运算符作用于输入流。因为 `istream` 本身并没有定义 `<<`，所以本来代码应该产生错误。然而，该代码能使用 `istream` 的 `bool` 类型转换运算符将 `cin` 转换成 `bool`，而这个 `bool` 值接着会被提升成 `int` 并用作内置的左移运算符的左侧运算对象。这样一来，提升后的 `bool` 值（1 或 0）最终会被左移 42 个位置。这一结果显然与我们的预期大相径庭。

显式的类型转换运算符

C++
11

为了防止这样的异常情况发生，C++11 新标准引入了显式的类型转换运算符（explicit conversion operator）：

```
class SmallInt {
public:
    // 编译器不会自动执行这一类型转换
    explicit operator int() const { return val; }
    // 其他成员与之前的版本一致
};
```

和显式的构造函数（参见 7.5.4 节，第 265 页）一样，编译器（通常）也不会将一个显式的类型转换运算符用于隐式类型转换：

```
SmallInt si = 3; // 正确：SmallInt 的构造函数不是显式的
si + 3; // 错误：此处需要隐式的类型转换，但类的运算符是显式的
static_cast<int>(si) + 3; // 正确：显式地请求类型转换
```

当类型转换运算符是显式的时，我们也能执行类型转换，不过必须通过显式的强制类型转换才可以。

该规定存在一个例外，即如果表达式被用作条件，则编译器会将显式的类型转换自动应用于它。换句话说，当表达式出现在下列位置时，显式的类型转换将被隐式地执行：

- `if`、`while` 及 `do` 语句的条件部分
- `for` 语句头的条件表达式
- 逻辑非运算符（`!`）、逻辑或运算符（`||`）、逻辑与运算符（`&&`）的运算对象
- 条件运算符（`? :`）的条件表达式。

583

转换为 `bool`

在标准库的早期版本中，IO 类型定义了向 `void*` 的转换规则，以求避免上面提到的问题。在 C++11 新标准下，IO 标准库通过定义一个向 `bool` 的显式类型转换实现同样的目的。

无论我们什么时候在条件中使用流对象，都会使用为 IO 类型定义的 `operator bool`。例如：

```
while (std::cin >> value)
```

`while` 语句的条件执行输入运算符，它负责将数据读入到 `value` 并返回 `cin`。为了对条件求值，`cin` 被 `istream` `operator bool` 类型转换函数隐式地执行了转换。如果 `cin` 的条件状态是 `good`（参见 8.1.2 节，第 280 页），则该函数返回为真；否则该函数返回为假。



向 `bool` 的类型转换通常用在条件部分，因此 `operator bool` 一般定义成 `explicit` 的。

14.9.1 节练习

练习 14.45: 编写类型转换运算符将一个 `Sales_data` 对象分别转换成 `string` 和 `double`，你认为这些运算符的返回值应该是什么？

练习 14.46: 你认为应该为 `Sales_data` 类定义上面两种类型转换运算符吗？应该把它们声明成 `explicit` 的吗？为什么？

练习 14.47: 说明下面这两个类型转换运算符的区别。

```
struct Integral {
    operator const int();
    operator int() const;
};
```

练习 14.48: 你在 7.5.1 节的练习 7.40 (第 261 页) 中曾经选择并编写了一个类，你认为它应该含有向 `bool` 的类型转换运算符吗？如果是，解释原因并说明该运算符是否应该是 `explicit` 的；如果不是，也请解释原因。

练习 14.49: 为上一题提到的类定义一个转换目标是 `bool` 的类型转换运算符，先不用在意这么做是否应该。

14.9.2 避免有二义性的类型转换



如果类中包含一个或多个类型转换，则必须确保在类类型和目标类型之间只存在唯一一种转换方式。否则的话，我们编写的代码将很可能具有二义性。

在两种情况下可能产生多重转换路径。第一种情况是两个类提供相同的类型转换：例如，当 `A` 类定义了一个接受 `B` 类对象的转换构造函数，同时 `B` 类定义了一个转换目标是 `A` 类的类型转换运算符时，我们就说它们提供了相同的类型转换。

第二种情况是类定义了多个转换规则，而这些转换涉及的类型本身可以通过其他类型转换联系在一起。最典型的例子是算术运算符，对某个给定的类来说，最好只定义最多一个与算术类型有关的转换规则。

◀ 584



通常情况下，不要为类定义相同的类型转换，也不要类中定义两个及两个以上转换源或转换目标是算术类型的转换。

实参匹配和相同的类型转换

在下面的例子中，我们定义了两种将 `B` 转换成 `A` 的方法：一种使用 `B` 的类型转换运算符、另一种使用 `A` 的以 `B` 为参数的构造函数：

```
// 最好不要在两个类之间构建相同的类型转换
struct B;
struct A {
    A() = default;
    A(const B&);           // 把一个 B 转换成 A
    // 其他数据成员
```

```
};
struct B {
    operator A() const; // 也是把一个 B 转换成 A
    // 其他数据成员
};
A f(const A&);
B b;
A a = f(b); // 二义性错误: 含义是 f(B::operator A())
            // 还是 f(A::A(const B&))?
```

因为同时存在两种由 B 获得 A 的方法, 所以造成编译器无法判断应该运行哪个类型转换, 也就是说, 对 f 的调用存在二义性。该调用可以使用以 B 为参数的 A 的构造函数, 也可以使用 B 当中把 B 转换成 A 的类型转换运算符。因为这两个函数效果相当、难分伯仲, 所以该调用将产生错误。

如果我们确实想执行上述的调用, 就不得不显式地调用类型转换运算符或者转换构造函数:

```
A a1 = f(b.operator A()); // 正确: 使用 B 的类型转换运算符
A a2 = f(A(b));           // 正确: 使用 A 的构造函数
```

值得注意的是, 我们无法使用强制类型转换来解决二义性问题, 因为强制类型转换本身也面临二义性。

二义性与转换目标为内置类型的多重类型转换

另外如果类定义了一组类型转换, 它们的转换源 (或者转换目标) 类型本身可以通过其他类型转换联系在一起, 则同样会产生二义性的问题。最简单也是最困扰我们的例子就是类当中定义了多个参数都是算术类型的构造函数, 或者转换目标都是算术类型的类型转换运算符。

例如, 在下面的类中包含两个转换构造函数, 它们的参数是两种不同的算术类型; 同时还包含两个类型转换运算符, 它们的转换目标也恰好是两种不同的算术类型:

585

```
struct A {
    A(int = 0);           // 最好不要创建两个转换源都是算术类型的类型转换
    A(double);
    operator int() const; // 最好不要创建两个转换对象都是算术类型的类型转换
    operator double() const;
    // 其他成员
};
void f2(long double);
A a;
f2(a); // 二义性错误: 含义是 f(A::operator int())
        // 还是 f(A::operator double())?

long lg;
A a2(lg); // 二义性错误: 含义是 A::A(int) 还是 A::A(double)?
```

在对 f2 的调用中, 哪个类型转换都无法精确匹配 long double, 然而这两个类型转换都可以使用, 只要后面再执行一次生成 long double 的标准类型转换即可。因此, 在上面的两个类型转换中哪个都不比另一个更好, 调用将产生二义性。