```
    b1 = b2;
    b2.push_back("about");
}
```

练习 12.2：编写你自己的 StrBlob 类，包含 const 版本的 front 和 back。

练习 12.3：StrBlob 需要 const 版本的 push_back 和 pop_back 吗？如果需要，添加进去。否则，解释为什么不需要。

练习 12.4：在我们的 check 函数中，没有检查 i 是否大于 0。为什么可以忽略这个检查？

练习 12.5：我们未编写接受一个 initializer_list explicit（参见 7.5.4 节，第 264 页）参数的构造函数。讨论这个设计策略的优点和缺点。

12.1.2 直接管理内存

C++语言定义了两个运算符来分配和释放动态内存。运算符 new 分配内存，delete 释放 new 分配的内存。

相对于智能指针，使用这两个运算符管理内存非常容易出错，随着我们逐步详细介绍这两个运算符，这一点会更为清楚。而且，自己直接管理内存的类与使用智能指针的类不同，它们不能依赖类对象拷贝、赋值和销毁操作的任何默认定义（参见 7.1.4 节，第 237 页）。因此，使用智能指针的程序更容易编写和调试。

在学习第 13 章之前，除非使用智能指针来管理内存，否则不要分配动态内存。

使用 new 动态分配和初始化对象

在自由空间分配的内存是无名的，因此 **new** 无法为其分配的对象命名，而是返回一个指向该对象的指针：

```
int *pi = new int; // pi 指向一个动态分配的、未初始化的无名对象
```

此 new 表达式在自由空间构造一个 int 型对象，并返回指向该对象的指针。

默认情况下，动态分配的对象是默认初始化的（参见 2.2.1 节，第 40 页），这意味着内置类型或组合类型的对象的值将是未定义的，而类类型对象将用默认构造函数进行初始化：

```
string *ps = new string; // 初始化为空 string
int *pi = new int;       // pi 指向一个未初始化的 int
```

459

我们可以使用直接初始化方式（参见 3.2.1 节，第 76 页）来初始化一个动态分配的对象。我们可以使用传统的构造方式（使用圆括号），在新标准下，也可以使用列表初始化（使用花括号）：

C++ 11

```
int *pi = new int(1024);              // pi 指向的对象的值为 1024
string *ps = new string(10, '9');     // *ps 为"9999999999"
// vector 有 10 个元素，值依次从 0 到 9

vector<int> *pv = new vector<int>{0,1,2,3,4,5,6,7,8,9};
```

也可以对动态分配的对象进行值初始化（参见 3.3.1 节，第 88 页），只需在类型名之

后跟一对空括号即可：

```
string *ps1 = new string;      // 默认初始化为空 string
string *ps = new string();     // 值初始化为空 string
int *pi1 = new int;            // 默认初始化；*pi1 的值未定义
int *pi2 = new int();          // 值初始化为 0；*pi2 为 0
```

对于定义了自己的构造函数（参见 7.1.4 节，第 235 页）的类类型（例如 string）来说，要求值初始化是没有意义的；不管采用什么形式，对象都会通过默认构造函数来初始化。但对于内置类型，两种形式的差别就很大了：值初始化的内置类型对象有着良好定义的值，而默认初始化的对象的值则是未定义的。类似的，对于类中那些依赖于编译器合成的默认构造函数的内置类型成员，如果它们未在类内被初始化，那么它们的值也是未定义的（参见 7.1.4 节，第 236 页）。

Best Practices

出于与变量初始化相同的原因，对动态分配的对象进行初始化通常是个好主意。

C++ 11

如果我们提供了一个括号包围的初始化器，就可以使用 auto（参见 2.5.2 节，第 61 页）从此初始化器来推断我们想要分配的对象的类型。但是，由于编译器要用初始化器的类型来推断要分配的类型，只有当括号中仅有单一初始化器时才可以使用 auto：

```
auto p1 = new auto(obj);       // p 指向一个与 obj 类型相同的对象
                               // 该对象用 obj 进行初始化
auto p2 = new auto{a,b,c};     // 错误：括号中只能有单个初始化器
```

p1 的类型是一个指针，指向从 obj 自动推断出的类型。若 obj 是一个 int，那么 p1 就是 int*；若 obj 是一个 string，那么 p1 是一个 string*；依此类推。新分配的对象用 obj 的值进行初始化。

动态分配的 const 对象

用 new 分配 const 对象是合法的：

```
// 分配并初始化一个 const int
const int *pci = new const int(1024);
// 分配并默认初始化一个 const 的空 string
const string *pcs = new const string;
```

460

类似其他任何 const 对象，一个动态分配的 const 对象必须进行初始化。对于一个定义了默认构造函数（参见 7.1.4 节，第 236 页）的类类型，其 const 动态对象可以隐式初始化，而其他类型的对象就必须显式初始化。由于分配的对象是 const 的，new 返回的指针是一个指向 const 的指针（参见 2.4.2 节，第 56 页）。

内存耗尽

虽然现代计算机通常都配备大容量内存，但是自由空间被耗尽的情况还是有可能发生。一旦一个程序用光了它所有可用的内存，new 表达式就会失败。默认情况下，如果 new 不能分配所要求的内存空间，它会抛出一个类型为 bad_alloc（参见 5.6 节，第 173 页）的异常。我们可以改变使用 new 的方式来阻止它抛出异常：

```
// 如果分配失败，new 返回一个空指针
int *p1 = new int; // 如果分配失败，new 抛出 std::bad_alloc
int *p2 = new (nothrow) int; // 如果分配失败，new 返回一个空指针
```

我们称这种形式的 new 为**定位 new**（placement new），其原因我们将在 19.1.2 节（第 729 页）中解释。定位 new 表达式允许我们向 new 传递额外的参数。在此例中，我们传递给它一个由标准库定义的名为 nothrow 的对象。如果将 nothrow 传递给 new，我们的意图是告诉它不能抛出异常。如果这种形式的 new 不能分配所需内存，它会返回一个空指针。bad_alloc 和 nothrow 都定义在头文件 new 中。

释放动态内存

为了防止内存耗尽，在动态内存使用完毕后，必须将其归还给系统。我们通过 **delete 表达式**（delete expression）来将动态内存归还给系统。delete 表达式接受一个指针，指向我们想要释放的对象：

```
delete p; // p 必须指向一个动态分配的对象或是一个空指针
```

与 new 类型类似，delete 表达式也执行两个动作：销毁给定的指针指向的对象；释放对应的内存。

指针值和 delete

我们传递给 delete 的指针必须指向动态分配的内存，或者是一个空指针（参见 2.3.2 节，第 48 页）。释放一块并非 new 分配的内存，或者将相同的指针值释放多次，其行为是未定义的：

```
int i, *pi1 = &i, *pi2 = nullptr;
double *pd = new double(33), *pd2 = pd;
delete i;    // 错误：i 不是一个指针
delete pi1;  // 未定义：pi1 指向一个局部变量
delete pd;   // 正确
delete pd2;  // 未定义：pd2 指向的内存已经被释放了
delete pi2;  // 正确：释放一个空指针总是没有错误的
```

对于 delete i 的请求，编译器会生成一个错误信息，因为它知道 i 不是一个指针。执行 delete pi1 和 pd2 所产生的错误则更具潜在危害：通常情况下，编译器不能分辨一个指针指向的是静态还是动态分配的对象。类似的，编译器也不能分辨一个指针所指向的内存是否已经被释放了。对于这些 delete 表达式，大多数编译器会编译通过，尽管它们是错误的。 461

虽然一个 const 对象的值不能被改变，但它本身是可以被销毁的。如同任何其他动态对象一样，想要释放一个 const 动态对象，只要 delete 指向它的指针即可：

```
const int *pci = new const int(1024);
delete pci; // 正确：释放一个 const 对象
```

动态对象的生存期直到被释放时为止

如 12.1.1 节（第 402 页）所述，由 shared_ptr 管理的内存在最后一个 shared_ptr 销毁时会被自动释放。但对于通过内置指针类型来管理的内存，就不是这样了。对于一个由内置指针管理的动态对象，直到被显式释放之前它都是存在的。

返回指向动态内存的指针（而不是智能指针）的函数给其调用者增加了一个额外负担——调用者必须记得释放内存：

```
// factory 返回一个指针，指向一个动态分配的对象
Foo* factory(T arg)
```

```
{
    // 视情况处理 arg
    return new Foo(arg); // 调用者负责释放此内存
}
```

类似我们之前定义的 factory 函数（参见 12.1.1 节，第 403 页），这个版本的 factory 分配一个对象，但并不 delete 它。factory 的调用者负责在不需要此对象时释放它。不幸的是，调用者经常忘记释放对象：

```
void use_factory(T arg)
{
    Foo *p = factory(arg);
    // 使用 p 但不 delete 它
} // p 离开了它的作用域，但它所指向的内存没有被释放！
```

此处，use_factory 函数调用 factory，后者分配一个类型为 Foo 的新对象。当 use_factory 返回时，局部变量 p 被销毁。此变量是一个内置指针，而不是一个智能指针。

与类类型不同，内置类型的对象被销毁时什么也不会发生。特别是，当一个指针离开其作用域时，它所指向的对象什么也不会发生。如果这个指针指向的是动态内存，那么内存将不会被自动释放。

由内置指针（而不是智能指针）管理的动态内存在被显式释放前一直都会存在。

462

在本例中，p 是指向 factory 分配的内存的唯一指针。一旦 use_factory 返回，程序就没有办法释放这块内存了。根据整个程序的逻辑，修正这个错误的正确方法是在 use_factory 中记得释放内存：

```
void use_factory(T arg)
{
    Foo *p = factory(arg);
    // 使用 p
    delete p; // 现在记得释放内存，我们已经不需要它了
}
```

还有一种可能，我们的系统中的其他代码要使用 use_factory 所分配的对象，我们就应该修改此函数，让它返回一个指针，指向它分配的内存：

```
Foo* use_factory(T arg)
{
    Foo *p = factory(arg);
    // 使用 p
    return p; // 调用者必须释放内存
}
```

小心：动态内存的管理非常容易出错

使用 new 和 delete 管理动态内存存在三个常见问题：

1. 忘记 delete 内存。忘记释放动态内存会导致人们常说的“内存泄漏”问题，因为这种内存永远不可能被归还给自由空间了。查找内存泄露错误是非常困难的，因为通常应用程序运行很长时间后，真正耗尽内存时，才能检测到这种错误。

2. 使用已经释放掉的对象。通过在释放内存后将指针置为空，有时可以检测出这

种错误。

3. 同一块内存释放两次。当有两个指针指向相同的动态分配对象时，可能发生这种错误。如果对其中一个指针进行了 delete 操作，对象的内存就被归还给自由空间了。如果我们随后又 delete 第二个指针，自由空间就可能被破坏。

相对于查找和修正这些错误来说，制造出这些错误要简单得多。

Best Practices 坚持只使用智能指针，就可以避免所有这些问题。对于一块内存，只有在没有任何智能指针指向它的情况下，智能指针才会自动释放它。

delete 之后重置指针值……

当我们 delete 一个指针后，指针值就变为无效了。虽然指针已经无效，但在很多机器上指针仍然保存着（已经释放了的）动态内存的地址。在 delete 之后，指针就变成了人们所说的**空悬指针**（dangling pointer），即，指向一块曾经保存数据对象但现在已经无效的内存的指针。 463

未初始化指针（参见 2.3.2 节，第 49 页）的所有缺点空悬指针也都有。有一种方法可以避免空悬指针的问题：在指针即将要离开其作用域之前释放掉它所关联的内存。这样，在指针关联的内存被释放掉之后，就没有机会继续使用指针了。如果我们需要保留指针，可以在 delete 之后将 nullptr 赋予指针，这样就清楚地指出指针不指向任何对象。

……这只是提供了有限的保护

动态内存的一个基本问题是可能有多个指针指向相同的内存。在 delete 内存之后重置指针的方法只对这个指针有效，对其他任何仍指向（已释放的）内存的指针是没有作用的。例如：

```
int *p(new int(42)); // p 指向动态内存
auto q = p;          // p 和 q 指向相同的内存
delete p;            // p 和 q 均变为无效
p = nullptr;         // 指出 p 不再绑定到任何对象
```

本例中 p 和 q 指向相同的动态分配的对象。我们 delete 此内存，然后将 p 置为 nullptr，指出它不再指向任何对象。但是，重置 p 对 q 没有任何作用，在我们释放 p 所指向的（同时也是 q 所指向的！）内存时，q 也变为无效了。在实际系统中，查找指向相同内存的所有指针是异常困难的。

12.1.2 节练习

练习 12.6：编写函数，返回一个动态分配的 int 的 vector。将此 vector 传递给另一个函数，这个函数读取标准输入，将读入的值保存在 vector 元素中。再将 vector 传递给另一个函数，打印读入的值。记得在恰当的时刻 delete vector。

练习 12.7：重做上一题，这次使用 shared_ptr 而不是内置指针。

练习 12.8：下面的函数是否有错误？如果有，解释错误原因。

```
bool b() {
    int* p = new int;
    // ...
```

```
    return p;
}
```

练习 12.9：解释下面代码执行的结果：

```
int *q = new int(42), *r = new int(100);
r = q;
auto q2 = make_shared<int>(42), r2 = make_shared<int>(100);
r2 = q2;
```

464

12.1.3 shared_ptr 和 new 结合使用

如前所述，如果我们不初始化一个智能指针，它就会被初始化为一个空指针。如表 12.3 所示，我们还可以用 new 返回的指针来初始化智能指针：

```
shared_ptr<double> p1; // shared_ptr 可以指向一个 double
shared_ptr<int> p2(new int(42)); // p2 指向一个值为 42 的 int
```

接受指针参数的智能指针构造函数是 explicit 的（参见 7.5.4 节，第 265 页）。因此，我们不能将一个内置指针隐式转换为一个智能指针，必须使用直接初始化形式（参见 3.2.1 节，第 76 页）来初始化一个智能指针：

```
shared_ptr<int> p1 = new int(1024);   // 错误：必须使用直接初始化形式
shared_ptr<int> p2(new int(1024));    // 正确：使用了直接初始化形式
```

p1 的初始化隐式地要求编译器用一个 new 返回的 int*来创建一个 shared_ptr。由于我们不能进行内置指针到智能指针间的隐式转换，因此这条初始化语句是错误的。出于相同的原因，一个返回 shared_ptr 的函数不能在其返回语句中隐式转换一个普通指针：

```
shared_ptr<int> clone(int p) {
    return new int(p); // 错误：隐式转换为 shared_ptr<int>
}
```

我们必须将 shared_ptr 显式绑定到一个想要返回的指针上：

```
shared_ptr<int> clone(int p) {
    //正确：显式地用 int*创建 shared_ptr<int>
    return shared_ptr<int>(new int(p));
}
```

默认情况下，一个用来初始化智能指针的普通指针必须指向动态内存，因为智能指针默认使用 delete 释放它所关联的对象。我们可以将智能指针绑定到一个指向其他类型的资源的指针上，但是为了这样做，必须提供自己的操作来替代 delete。我们将在 12.1.4 节（第 415 页）介绍如何定义自己的释放操作。

表 12.3：定义和改变 shared_ptr 的其他方法

shared_ptr<T> p(q)	p 管理内置指针 q 所指向的对象；q 必须指向 new 分配的内存，且能够转换为 T*类型
shared_ptr<T> p(u)	p 从 unique_ptr u 那里接管了对象的所有权；将 u 置为空
shared_ptr<T> p(q, d)	p 接管了内置指针 q 所指向的对象的所有权。q 必须能转换为 T*类型（参见 4.11.2 节，第 143 页）。p 将使用可调用对象 d（参见 10.3.2 节，第 346 页）来代替 delete

续表

shared_ptr<T> p(p2, d)	如表 12.2 所示，p 是 shared_ptr p2 的拷贝，唯一的区别是 p 将用可调用对象 d 来代替 delete
p.reset() p.reset(q) p.reset(q, d)	若 p 是唯一指向其对象的 shared_ptr，reset 会释放此对象。若传递了可选的参数内置指针 q，会令 p 指向 q，否则会将 p 置为空。若还传递了参数 d，将会调用 d 而不是 delete 来释放 q

不要混合使用普通指针和智能指针……

shared_ptr 可以协调对象的析构，但这仅限于其自身的拷贝（也是 shared_ptr）之间。这也是为什么我们推荐使用 make_shared 而不是 new 的原因。这样，我们就能在分配对象的同时就将 shared_ptr 与之绑定，从而避免了无意中将同一块内存绑定到多个独立创建的 shared_ptr 上。

考虑下面对 shared_ptr 进行操作的函数：

```
// 在函数被调用时 ptr 被创建并初始化
void process(shared_ptr<int> ptr)
{
    // 使用 ptr
} // ptr 离开作用域，被销毁
```

process 的参数是传值方式传递的，因此实参会被拷贝到 ptr 中。拷贝一个 shared_ptr 会递增其引用计数，因此，在 process 运行过程中，引用计数值至少为 2。当 process 结束时，ptr 的引用计数会递减，但不会变为 0。因此，当局部变量 ptr 被销毁时，ptr 指向的内存不会被释放。 465

使用此函数的正确方法是传递给它一个 shared_ptr：

```
shared_ptr<int> p(new int(42)); // 引用计数为 1
process(p); // 拷贝 p 会递增它的引用计数；在 process 中引用计数值为 2
int i = *p; // 正确：引用计数值为 1
```

虽然不能传递给 process 一个内置指针，但可以传递给它一个（临时的）shared_ptr，这个 shared_ptr 是用一个内置指针显式构造的。但是，这样做很可能会导致错误：

```
int *x(new int(1024));        // 危险：x 是一个普通指针，不是一个智能指针
process(x); // 错误：不能将 int*转换为一个 shared_ptr<int>
process(shared_ptr<int>(x)); // 合法的，但内存会被释放！
int j = *x; // 未定义的：x 是一个空悬指针！
```

在上面的调用中，我们将一个临时 shared_ptr 传递给 process。当这个调用所在的表达式结束时，这个临时对象就被销毁了。销毁这个临时变量会递减引用计数，此时引用计数就变为 0 了。因此，当临时对象被销毁时，它所指向的内存会被释放。

但 x 继续指向（已经释放的）内存，从而变成一个空悬指针。如果试图使用 x 的值，其行为是未定义的。

当将一个 shared_ptr 绑定到一个普通指针时，我们就将内存的管理责任交给了这个 shared_ptr。一旦这样做了，我们就不应该再使用内置指针来访问 shared_ptr 所指向的内存了。

466

WARNING

使用一个内置指针来访问一个智能指针所负责的对象是很危险的，因为我们无法知道对象何时会被销毁。

……也不要使用 get 初始化另一个智能指针或为智能指针赋值

智能指针类型定义了一个名为 get 的函数（参见表 12.1），它返回一个内置指针，指向智能指针管理的对象。此函数是为了这样一种情况而设计的：我们需要向不能使用智能指针的代码传递一个内置指针。使用 get 返回的指针的代码不能 delete 此指针。

虽然编译器不会给出错误信息，但将另一个智能指针也绑定到 get 返回的指针上是错误的：

```
shared_ptr<int> p(new int(42)); // 引用计数为 1
int *q = p.get(); // 正确：但使用 q 时要注意，不要让它管理的指针被释放
{ // 新程序块
// 未定义：两个独立的 shared_ptr 指向相同的内存
shared_ptr<int>(q);
} // 程序块结束，q 被销毁，它指向的内存被释放
int foo = *p; // 未定义：p 指向的内存已经被释放了
```

在本例中，p 和 q 指向相同的内存。由于它们是相互独立创建的，因此各自的引用计数都是 1。当 q 所在的程序块结束时，q 被销毁，这会导致 q 指向的内存被释放。从而 p 变成一个空悬指针，意味着当我们试图使用 p 时，将发生未定义的行为。而且，当 p 被销毁时，这块内存会被第二次 delete。

WARNING

get 用来将指针的访问权限传递给代码，你只有在确定代码不会 delete 指针的情况下，才能使用 get。特别是，永远不要用 get 初始化另一个智能指针或者为另一个智能指针赋值。

其他 shared_ptr 操作

shared_ptr 还定义了其他一些操作，参见表 12.2 和表 12.3 所示。我们可以用 reset 来将一个新的指针赋予一个 shared_ptr：

```
p = new int(1024);          // 错误：不能将一个指针赋予 shared_ptr
p.reset(new int(1024));     // 正确：p 指向一个新对象
```

与赋值类似，reset 会更新引用计数，如果需要的话，会释放 p 指向的对象。reset 成员经常与 unique 一起使用，来控制多个 shared_ptr 共享的对象。在改变底层对象之前，我们检查自己是否是当前对象仅有的用户。如果不是，在改变之前要制作一份新的拷贝：

```
if (!p.unique())
    p.reset(new string(*p)); // 我们不是唯一用户；分配新的拷贝
*p += newVal; // 现在我们知道自己是唯一的用户，可以改变对象的值
```

467

12.1.3 节练习

练习 12.10：下面的代码调用了第 413 页中定义的 process 函数，解释此调用是否正确。如果不正确，应如何修改？

```
shared_ptr<int> p(new int(42));
```

```
process(shared_ptr<int>(p));
```

练习 12.11：如果我们像下面这样调用 process，会发生什么？

```
process(shared_ptr<int>(p.get()));
```

练习 12.12：p 和 q 的定义如下，对于接下来的对 process 的每个调用，如果合法，解释它做了什么，如果不合法，解释错误原因：

```
auto p = new int();
auto sp = make_shared<int>();
(a) process(sp);
(b) process(new int());
(c) process(p);
(d) process(shared_ptr<int>(p));
```

练习 12.13：如果执行下面的代码，会发生什么？

```
auto sp = make_shared<int>();
auto p = sp.get();
delete p;
```

12.1.4 智能指针和异常

5.6.2 节（第 175 页）中介绍了使用异常处理的程序能在异常发生后令程序流程继续，我们注意到，这种程序需要确保在异常发生后资源能被正确地释放。一个简单的确保资源被释放的方法是使用智能指针。

如果使用智能指针，即使程序块过早结束，智能指针类也能确保在内存不再需要时将其释放，：

```
void f()
{
    shared_ptr<int> sp(new int(42)); // 分配一个新对象
    //这段代码抛出一个异常，且在 f 中未被捕获
}  // 在函数结束时 shared_ptr 自动释放内存
```

函数的退出有两种可能，正常处理结束或者发生了异常，无论哪种情况，局部对象都会被销毁。在上面的程序中，sp 是一个 shared_ptr，因此 sp 销毁时会检查引用计数。在此例中，sp 是指向这块内存的唯一指针，因此内存会被释放掉。

与之相对的，当发生异常时，我们直接管理的内存是不会自动释放的。如果使用内置指针管理内存，且在 new 之后在对应的 delete 之前发生了异常，则内存不会被释放：

```
void f()
{
    int *ip = new int(42);    // 动态分配一个新对象
    // 这段代码抛出一个异常，且在 f 中未被捕获
    delete ip;                // 在退出之前释放内存
}
```

468

如果在 new 和 delete 之间发生异常，且异常未在 f 中被捕获，则内存就永远不会被释放了。在函数 f 之外没有指针指向这块内存，因此就无法释放它了。

智能指针和哑类

包括所有标准库类在内的很多 C++类都定义了析构函数（参见 12.1.1 节，第 402 页），负责清理对象使用的资源。但是，不是所有的类都是这样良好定义的。特别是那些为 C 和 C++两种语言设计的类，通常都要求用户显式地释放所使用的任何资源。

那些分配了资源，而又没有定义析构函数来释放这些资源的类，可能会遇到与使用动态内存相同的错误——程序员非常容易忘记释放资源。类似的，如果在资源分配和释放之间发生了异常，程序也会发生资源泄漏。

与管理动态内存类似，我们通常可以使用类似的技术来管理不具有良好定义的析构函数的类。例如，假定我们正在使用一个 C 和 C++都使用的网络库，使用这个库的代码可能是这样的：

```
struct destination;                    // 表示我们正在连接什么
struct connection;                     // 使用连接所需的信息
connection connect(destination*);      // 打开连接
void disconnect(connection);           // 关闭给定的连接
void f(destination &d /* 其他参数 */)
{
    // 获得一个连接；记住使用完后要关闭它
    connection c = connect(&d);
    // 使用连接
    // 如果我们在 f 退出前忘记调用 disconnect，就无法关闭 c 了
}
```

如果 connection 有一个析构函数，就可以在 f 结束时由析构函数自动关闭连接。但是，connection 没有析构函数。这个问题与我们上一个程序中使用 shared_ptr 避免内存泄漏几乎是等价的。使用 shared_ptr 来保证 connection 被正确关闭，已被证明是一种有效的方法。

使用我们自己的释放操作

469

默认情况下，shared_ptr 假定它们指向的是动态内存。因此，当一个 shared_ptr 被销毁时，它默认地对它管理的指针进行 delete 操作。为了用 shared_ptr 来管理一个 connection，我们必须首先定义一个函数来代替 delete。这个**删除器**（deleter）函数必须能够完成对 shared_ptr 中保存的指针进行释放的操作。在本例中，我们的删除器必须接受单个类型为 connection*的参数：

```
void end_connection(connection *p) { disconnect(*p); }
```

当我们创建一个 shared_ptr 时，可以传递一个（可选的）指向删除器函数的参数（参见 6.7 节，第 221 页）：

```
void f(destination &d /* 其他参数 */)
{
    connection c = connect(&d);
    shared_ptr<connection> p(&c, end_connection);
    // 使用连接
    // 当 f 退出时（即使是由于异常而退出），connection 会被正确关闭
}
```

当 p 被销毁时，它不会对自己保存的指针执行 delete，而是调用 end_connection。接下来，end_connection 会调用 disconnect，从而确保连接被关闭。如果 f 正常退出，那么 p 的销毁会作为结束处理的一部分。如果发生了异常，p 同样会被销毁，从而连接被关闭。

注意：智能指针陷阱

智能指针可以提供对动态分配的内存安全而又方便的管理，但这建立在正确使用的前提下。为了正确使用智能指针，我们必须坚持一些基本规范：

- 不使用相同的内置指针值初始化（或 reset）多个智能指针。
- 不 delete get()返回的指针。
- 不使用 get()初始化或 reset 另一个智能指针。
- 如果你使用 get()返回的指针，记住当最后一个对应的智能指针销毁后，你的指针就变为无效了。
- 如果你使用智能指针管理的资源不是 new 分配的内存，记住传递给它一个删除器（参见 12.1.4 节，第 415 页和 12.1.5 节，第 419 页）。

12.1.4 节练习

练习 12.14：编写你自己版本的用 shared_ptr 管理 connection 的函数。

练习 12.15：重写第一题的程序，用 lambda（参见 10.3.2 节，第 346 页）代替 end_connection 函数。

12.1.5 unique_ptr

470

一个 unique_ptr "拥有"它所指向的对象。与 shared_ptr 不同，某个时刻只能有一个 unique_ptr 指向一个给定对象。当 unique_ptr 被销毁时，它所指向的对象也被销毁。表 12.4 列出了 unique_ptr 特有的操作。与 shared_ptr 相同的操作列在表 12.1（第 401 页）中。

C++ 11

与 shared_ptr 不同，没有类似 make_shared 的标准库函数返回一个 unique_ptr。当我们定义一个 unique_ptr 时，需要将其绑定到一个 new 返回的指针上。类似 shared_ptr，初始化 unique_ptr 必须采用直接初始化形式：

```
unique_ptr<double> p1; // 可以指向一个 double 的 unique_ptr
unique_ptr<int> p2(new int(42)); // p2 指向一个值为 42 的 int
```

由于一个 unique_ptr 拥有它指向的对象，因此 unique_ptr 不支持普通的拷贝或赋值操作：

```
unique_ptr<string> p1(new string("Stegosaurus"));
unique_ptr<string> p2(p1);    // 错误：unique_ptr 不支持拷贝
unique_ptr<string> p3;
p3 = p2;                      // 错误：unique_ptr 不支持赋值
```

表 12.4：unique_ptr 操作（另参见表 12.1，第 401 页）	
unique_ptr<T> u1 unique_ptr<T, D> u2	空 unique_ptr，可以指向类型为 T 的对象。u1 会使用 delete 来释放它的指针；u2 会使用一个类型为 D 的可调用对象来释放它的指针
unique_ptr<T, D> u(d)	空 unique_ptr，指向类型为 T 的对象，用类型为 D 的对象 d 代替 delete
u = nullptr	释放 u 指向的对象，将 u 置为空
u.release()	u 放弃对指针的控制权，返回指针，并将 u 置为空
u.reset()	释放 u 指向的对象
u.reset(q) u.reset(nullptr)	如果提供了内置指针 q，令 u 指向这个对象；否则将 u 置为空

虽然我们不能拷贝或赋值 unique_ptr，但可以通过调用 release 或 reset 将指针的所有权从一个（非 const）unique_ptr 转移给另一个 unique：

```
// 将所有权从 p1（指向 string Stegosaurus）转移给 p2
unique_ptr<string> p2(p1.release()); // release 将 p1 置为空
unique_ptr<string> p3(new string("Trex"));
// 将所有权从 p3 转移给 p2
p2.reset(p3.release()); // reset 释放了 p2 原来指向的内存
```

release 成员返回 unique_ptr 当前保存的指针并将其置为空。因此，p2 被初始化为 p1 原来保存的指针，而 p1 被置为空。

471 reset 成员接受一个可选的指针参数，令 unique_ptr 重新指向给定的指针。如果 unique_ptr 不为空，它原来指向的对象被释放。因此，对 p2 调用 reset 释放了用 "Stegosaurus"初始化的 string 所使用的内存，将 p3 对指针的所有权转移给 p2，并将 p3 置为空。

调用 release 会切断 unique_ptr 和它原来管理的对象间的联系。release 返回的指针通常被用来初始化另一个智能指针或给另一个智能指针赋值。在本例中，管理内存的责任简单地从一个智能指针转移给另一个。但是，如果我们不用另一个智能指针来保存 release 返回的指针，我们的程序就要负责资源的释放：

```
p2.release();          // 错误：p2 不会释放内存，而且我们丢失了指针
auto p = p2.release();  // 正确，但我们必须记得 delete(p)
```

传递 unique_ptr 参数和返回 unique_ptr

不能拷贝 unique_ptr 的规则有一个例外：我们可以拷贝或赋值一个将要被销毁的 unique_ptr。最常见的例子是从函数返回一个 unique_ptr：

```
unique_ptr<int> clone(int p) {
    // 正确：从 int*创建一个 unique_ptr<int>
    return unique_ptr<int>(new int(p));
}
```

还可以返回一个局部对象的拷贝：

```
unique_ptr<int> clone(int p) {
    unique_ptr<int> ret(new int (p));
    // ...
    return ret;
}
```

对于两段代码，编译器都知道要返回的对象将要被销毁。在此情况下，编译器执行一种特殊的“拷贝”，我们将在 13.6.2 节（第 473 页）中介绍它。

向后兼容：auto_ptr

标准库的较早版本包含了一个名为 auto_ptr 的类，它具有 unique_ptr 的部分特性，但不是全部。特别是，我们不能在容器中保存 auto_ptr，也不能从函数中返回 auto_ptr。

虽然 auto_ptr 仍是标准库的一部分，但编写程序时应该使用 unique_ptr。

向 unique_ptr 传递删除器

类似 shared_ptr，unique_ptr 默认情况下用 delete 释放它指向的对象。与 shared_ptr 一样，我们可以重载一个 unique_ptr 中默认的删除器（参见 12.1.4 节，第 415 页）。但是，unique_ptr 管理删除器的方式与 shared_ptr 不同，其原因我们将在 16.1.6 节（第 599 页）中介绍。 472

重载一个 unique_ptr 中的删除器会影响到 unique_ptr 类型以及如何构造（或 reset）该类型的对象。与重载关联容器的比较操作（参见 11.2.2 节，第 378 页）类似，我们必须在尖括号中 unique_ptr 指向类型之后提供删除器类型。在创建或 reset 一个这种 unique_ptr 类型的对象时，必须提供一个指定类型的可调用对象（删除器）：

```
// p 指向一个类型为 objT 的对象，并使用一个类型为 delT 的对象释放 objT 对象
// 它会调用一个名为 fcn 的 delT 类型对象
unique_ptr<objT, delT> p (new objT, fcn);
```

作为一个更具体的例子，我们将重写连接程序，用 unique_ptr 来代替 shared_ptr，如下所示：

```
void f(destination &d /* 其他需要的参数 */)
{
    connection c = connect(&d); // 打开连接
    // 当 p 被销毁时，连接将会关闭
    unique_ptr<connection, decltype(end_connection)*>
        p(&c, end_connection);
    // 使用连接
    // 当 f 退出时（即使是由于异常而退出），connection 会被正确关闭
}
```

在本例中我们使用了 decltype（参见 2.5.3 节，第 62 页）来指明函数指针类型。由于 decltype(end_connection) 返回一个函数类型，所以我们必须添加一个*来指出我们正在使用该类型的一个指针（参见 6.7 节，第 223 页）。

12.1.5 节练习

练习 12.16：如果你试图拷贝或赋值 unique_ptr，编译器并不总是能给出易于理解的错误信息。编写包含这种错误的程序，观察编译器如何诊断这种错误。

练习 12.17：下面的 unique_ptr 声明中，哪些是合法的，哪些可能导致后续的程序错误？解释每个错误的问题在哪里。

```
int ix = 1024, *pi = &ix, *pi2 = new int(2048);
typedef unique_ptr<int> IntP;
(a) IntP p0(ix);                    (b) IntP p1(pi);
(c) IntP p2(pi2);                   (d) IntP p3(&ix);
(e) IntP p4(new int(2048));         (f) IntP p5(p2.get());
```

练习 12.18：shared_ptr 为什么没有 release 成员？

12.1.6 weak_ptr

473 C++ 11

weak_ptr（见表 12.5）是一种不控制所指向对象生存期的智能指针，它指向由一个 shared_ptr 管理的对象。将一个 weak_ptr 绑定到一个 shared_ptr 不会改变 shared_ptr 的引用计数。一旦最后一个指向对象的 shared_ptr 被销毁，对象就会被释放。即使有 weak_ptr 指向对象，对象也还是会被释放，因此，weak_ptr 的名字抓住了这种智能指针“弱”共享对象的特点。

表 12.5：weak_ptr

weak_ptr<T> w	空 weak_ptr 可以指向类型为 T 的对象
weak_ptr<T> w(sp)	与 shared_ptr sp 指向相同对象的 weak_ptr。T 必须能转换为 sp 指向的类型
w = p	p 可以是一个 shared_ptr 或一个 weak_ptr。赋值后 w 与 p 共享对象
w.reset()	将 w 置为空
w.use_count()	与 w 共享对象的 shared_ptr 的数量
w.expired()	若 w.use_count() 为 0，返回 true，否则返回 false
w.lock()	如果 expired 为 true，返回一个空 shared_ptr；否则返回一个指向 w 的对象的 shared_ptr

当我们创建一个 weak_ptr 时，要用一个 shared_ptr 来初始化它：

```
auto p = make_shared<int>(42);
weak_ptr<int> wp(p); // wp 弱共享 p; p 的引用计数未改变
```

本例中 wp 和 p 指向相同的对象。由于是弱共享，创建 wp 不会改变 p 的引用计数；wp 指向的对象可能被释放掉。

由于对象可能不存在，我们不能使用 weak_ptr 直接访问对象，而必须调用 lock。此函数检查 weak_ptr 指向的对象是否仍存在。如果存在，lock 返回一个指向共享对象的 shared_ptr。与任何其他 shared_ptr 类似，只要此 shared_ptr 存在，它所指向的底层对象也就会一直存在。例如：

```
if (shared_ptr<int> np = wp.lock()) { // 如果 np 不为空则条件成立
    // 在 if 中，np 与 p 共享对象
}
```

在这段代码中，只有当 lock 调用返回 true 时我们才会进入 if 语句体。在 if 中，使用 np 访问共享对象是安全的。

核查指针类

作为 weak_ptr 用途的一个展示，我们将为 StrBlob 类定义一个伴随指针类。我们

的指针类将命名为 StrBlobPtr，会保存一个 weak_ptr，指向 StrBlob 的 data 成员，这是初始化时提供给它的。通过使用 weak_ptr，不会影响一个给定的 StrBlob 所指向的 vector 的生存期。但是，可以阻止用户访问一个不再存在的 vector 的企图。 474

StrBlobPtr 会有两个数据成员：wptr，或者为空，或者指向一个 StrBlob 中的 vector；curr，保存当前对象所表示的元素的下标。类似它的伴随类 StrBlob，我们的指针类也有一个 check 成员来检查解引用 StrBlobPtr 是否安全：

```
// 对于访问一个不存在元素的尝试，StrBlobPtr 抛出一个异常
class StrBlobPtr {
public:
    StrBlobPtr(): curr(0) { }
    StrBlobPtr(StrBlob &a, size_t sz = 0):
            wptr(a.data), curr(sz) { }
    std::string& deref() const;
    StrBlobPtr& incr(); // 前缀递增
private:
    // 若检查成功，check 返回一个指向 vector 的 shared_ptr
    std::shared_ptr<std::vector<std::string>>
        check(std::size_t, const std::string&) const;
    // 保存一个 weak_ptr，意味着底层 vector 可能会被销毁
    std::weak_ptr<std::vector<std::string>> wptr;
    std::size_t curr; // 在数组中的当前位置
};
```

默认构造函数生成一个空的 StrBlobPtr。其构造函数初始化列表（参见 7.1.4 节，第 237 页）将 curr 显式初始化为 0，并将 wptr 隐式初始化为一个空 weak_ptr。第二个构造函数接受一个 StrBlob 引用和一个可选的索引值。此构造函数初始化 wptr，令其指向给定 StrBlob 对象的 shared_ptr 中的 vector，并将 curr 初始化为 sz 的值。我们使用了默认参数（参见 6.5.1 节，第 211 页），表示默认情况下将 curr 初始化为第一个元素的下标。我们将会看到，StrBlob 的 end 成员将会用到参数 sz。

值得注意的是，我们不能将 StrBlobPtr 绑定到一个 const StrBlob 对象。这个限制是由于构造函数接受一个非 const StrBlob 对象的引用而导致的。

StrBlobPtr 的 check 成员与 StrBlob 中的同名成员不同，它还要检查指针指向的 vector 是否还存在：

```
std::shared_ptr<std::vector<std::string>>
StrBlobPtr::check(std::size_t i, const std::string &msg) const
{
    auto ret = wptr.lock(); // vector 还存在吗?
    if (!ret)
        throw std::runtime_error("unbound StrBlobPtr");
    if (i >= ret->size())
        throw std::out_of_range(msg);
    return ret; // 否则，返回指向 vector 的 shared_ptr
}
```

由于一个 weak_ptr 不参与其对应的 shared_ptr 的引用计数，StrBlobPtr 指向的 vector 可能已经被释放了。如果 vector 已销毁，lock 将返回一个空指针。在本例中，任何 vector 的引用都会失败，于是抛出一个异常。否则，check 会检查给定索引，如果索引值合法，check 返回从 lock 获得的 shared_ptr。 475

指针操作

我们将在第 14 章学习如何定义自己的运算符。现在，我们将定义名为 deref 和 incr 的函数，分别用来解引用和递增 StrBlobPtr。

deref 成员调用 check，检查使用 vector 是否安全以及 curr 是否在合法范围内：

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

如果 check 成功，p 就是一个 shared_ptr，指向 StrBlobPtr 所指向的 vector。表达式 (*p) [curr] 解引用 shared_ptr 来获得 vector，然后使用下标运算符提取并返回 curr 位置上的元素。

incr 成员也调用 check：

```
// 前缀递增：返回递增后的对象的引用
StrBlobPtr& StrBlobPtr::incr()
{
    // 如果 curr 已经指向容器的尾后位置，就不能递增它
    check(curr, "increment past end of StrBlobPtr");
    ++curr; // 推进当前位置
    return *this;
}
```

当然，为了访问 data 成员，我们的指针类必须声明为 StrBlob 的 friend（参见 7.3.4 节，第 250 页）。我们还要为 StrBlob 类定义 begin 和 end 操作，返回一个指向它自身的 StrBlobPtr：

```
// 对于 StrBlob 中的友元声明来说，此前置声明是必要的
class StrBlobPtr;
class StrBlob {
    friend class StrBlobPtr;
    // 其他成员与 12.1.1 节（第 405 页）中声明相同
    // 返回指向首元素和尾后元素的 StrBlobPtr
    StrBlobPtr begin() { return StrBlobPtr(*this); }
    StrBlobPtr end()
        { auto ret = StrBlobPtr(*this, data->size());
          return ret; }
};
```

476

12.1.6 节练习

练习 12.19：定义你自己版本的 StrBlobPtr，更新 StrBlob 类，加入恰当的 friend 声明及 begin 和 end 成员。

练习 12.20：编写程序，逐行读入一个输入文件，将内容存入一个 StrBlob 中，用一个 StrBlobPtr 打印出 StrBlob 中的每个元素。

练习 12.21：也可以这样编写 StrBlobPtr 的 deref 成员：

```
std::string& deref() const
```

```
    { return (*check(curr, "dereference past end"))[curr]; }
```

你认为哪个版本更好？为什么？

练习 12.22：为了能让 StrBlobPtr 使用 const StrBlob，你觉得应该如何修改？定义一个名为 ConstStrBlobPtr 的类，使其能够指向 const StrBlob。

12.2 动态数组

new 和 delete 运算符一次分配/释放一个对象，但某些应用需要一次为很多对象分配内存的功能。例如，vector 和 string 都是在连续内存中保存它们的元素，因此，当容器需要重新分配内存时（参见 9.4 节，第 317 页），必须一次性为很多元素分配内存。

为了支持这种需求，C++语言和标准库提供了两种一次分配一个对象数组的方法。C++语言定义了另一种 new 表达式语法，可以分配并初始化一个对象数组。标准库中包含一个名为 allocator 的类，允许我们将分配和初始化分离。使用 allocator 通常会提供更好的性能和更灵活的内存管理能力，原因我们将在 12.2.2 节（第 427 页）中解释。

很多（可能是大多数）应用都没有直接访问动态数组的需求。当一个应用需要可变数量的对象时，我们在 StrBlob 中所采用的方法几乎总是更简单、更快速并且更安全的——即，使用 vector（或其他标准库容器）。如我们将在 13.6 节（第 470 页）中看到的，使用标准库容器的优势在新标准下更为显著。在支持新标准的标准库中，容器操作比之前的版本要快速得多。

Best Practices 大多数应用应该使用标准库容器而不是动态分配的数组。使用容器更为简单、更不容易出现内存管理错误并且可能有更好的性能。

如前所述，使用容器的类可以使用默认版本的拷贝、赋值和析构操作（参见 7.1.5 节，第 239 页）。分配动态数组的类则必须定义自己版本的操作，在拷贝、复制以及销毁对象时管理所关联的内存。

WARNING 直到学习完第 13 章，不要在类内的代码中分配动态内存。

12.2.1 new 和数组

为了让 new 分配一个对象数组，我们要在类型名之后跟一对方括号，在其中指明要分配的对象的数目。在下例中，new 分配要求数量的对象并（假定分配成功后）返回指向第一个对象的指针： 477

```
// 调用 get_size 确定分配多少个 int
int *pia = new int[get_size()]; // pia 指向第一个 int
```

方括号中的大小必须是整型，但不必是常量。

也可以用一个表示数组类型的类型别名（参见 2.5.1 节，第 60 页）来分配一个数组，这样，new 表达式中就不需要方括号了：

```
typedef int arrT[42];      // arrT 表示 42 个 int 的数组类型
int *p = new arrT;         // 分配一个 42 个 int 的数组；p 指向第一个 int
```

在本例中，new 分配一个 int 数组，并返回指向第一个 int 的指针。即使这段代码中没

有方括号，编译器执行这个表达式时还是会用 new[]。即，编译器执行如下形式：

```
int *p = new int[42];
```

分配一个数组会得到一个元素类型的指针

虽然我们通常称 new T[]分配的内存为“动态数组”，但这种叫法某种程度上有些误导。当用 new 分配一个数组时，我们并未得到一个数组类型的对象，而是得到一个数组元素类型的指针。即使我们使用类型别名定义了一个数组类型，new 也不会分配一个数组类型的对象。在上例中，我们正在分配一个数组的事实甚至都是不可见的——连[*num*]都没有。new 返回的是一个元素类型的指针。

由于分配的内存并不是一个数组类型，因此不能对动态数组调用 begin 或 end（参见 3.5.3 节，第 106 页）。这些函数使用数组维度（回忆一下，维度是数组类型的一部分）来返回指向首元素和尾后元素的指针。出于相同的原因，也不能用范围 for 语句来处理（所谓的）动态数组中的元素。

C++ 11

要记住我们所说的动态数组并不是数组类型，这是很重要的。

初始化动态分配对象的数组

默认情况下，new 分配的对象，不管是单个分配的还是数组中的，都是默认初始化的。可以对数组中的元素进行值初始化（参见 3.3.1 节，第 88 页），方法是在大小之后跟一对空括号。

```
int *pia = new int[10];           // 10 个未初始化的 int
int *pia2 = new int[10]();        // 10 个值初始化为 0 的 int
string *psa = new string[10];     // 10 个空 string
string *psa2 = new string[10]();  // 10 个空 string
```

478 在新标准中，我们还可以提供一个元素初始化器的花括号列表：

C++ 11

```
// 10 个 int 分别用列表中对应的初始化器初始化
int *pia3 = new int[10]{0,1,2,3,4,5,6,7,8,9};
// 10 个 string，前 4 个用给定的初始化器初始化，剩余的进行值初始化
string *psa3 = new string[10]{"a", "an", "the", string(3,'x')};
```

与内置数组对象的列表初始化（参见 3.5.1 节，第 102 页）一样，初始化器会用来初始化动态数组中开始部分的元素。如果初始化器数目小于元素数目，剩余元素将进行值初始化。如果初始化器数目大于元素数目，则 new 表达式失败，不会分配任何内存。在本例中，new 会抛出一个类型为 bad_array_new_length 的异常。类似 bad_alloc，此类型定义在头文件 new 中。

C++ 11

虽然我们用空括号对数组中元素进行值初始化，但不能在括号中给出初始化器，这意味着不能用 auto 分配数组（参见 12.1.2 节，第 407 页）。

动态分配一个空数组是合法的

可以用任意表达式来确定要分配的对象的数目：

```
size_t n = get_size();    // get_size 返回需要的元素的数目
int* p = new int[n];      // 分配数组保存元素
for (int* q = p; q != p + n; ++q)
    /* 处理数组 */ ;
```

这产生了一个有意思的问题：如果 get_size 返回 0，会发生什么？答案是代码仍能正常工作。虽然我们不能创建一个大小为 0 的静态数组对象，但当 n 等于 0 时，调用 new[n] 是合法的：

```
char arr[0];                // 错误：不能定义长度为 0 的数组
char *cp = new char[0];     // 正确：但 cp 不能解引用
```

当我们用 new 分配一个大小为 0 的数组时，new 返回一个合法的非空指针。此指针保证与 new 返回的其他任何指针都不相同。对于零长度的数组来说，此指针就像尾后指针一样（参见 3.5.3 节，第 106 页），我们可以像使用尾后迭代器一样使用这个指针。可以用此指针进行比较操作，就像上面循环代码中那样。可以向此指针加上（或从此指针减去）0，也可以从此指针减去自身从而得到 0。但此指针不能解引用——毕竟它不指向任何元素。

在我们假想的循环中，若 get_size 返回 0，则 n 也是 0，new 会分配 0 个对象。for 循环中的条件会失败（p 等于 q+n，因为 n 为 0）。因此，循环体不会被执行。

释放动态数组

为了释放动态数组，我们使用一种特殊形式的 delete——在指针前加上一个空方括号对：

```
delete p;          // p 必须指向一个动态分配的对象或为空
delete [] pa;      // pa 必须指向一个动态分配的数组或为空
```

479

第二条语句销毁 pa 指向的数组中的元素，并释放对应的内存。数组中的元素按逆序销毁，即，最后一个元素首先被销毁，然后是倒数第二个，依此类推。

当我们释放一个指向数组的指针时，空方括号对是必需的：它指示编译器此指针指向一个对象数组的第一个元素。如果我们在 delete 一个指向数组的指针时忽略了方括号（或者在 delete 一个指向单一对象的指针时使用了方括号），其行为是未定义的。

回忆一下，当我们使用一个类型别名来定义一个数组类型时，在 new 表达式中不使用[]。即使是这样，在释放一个数组指针时也必须使用方括号：

```
typedef int arrT[42];      // arrT 是 42 个 int 的数组的类型别名
int *p = new arrT;         // 分配一个 42 个 int 的数组；p 指向第一个元素
delete [] p;               // 方括号是必需的，因为我们当初分配的是一个数组
```

不管外表如何，p 指向一个对象数组的首元素，而不是一个类型为 arrT 的单一对象。因此，在释放 p 时我们必须使用[]。

WARNING

如果我们在 delete 一个数组指针时忘记了方括号，或者在 delete 一个单一对象的指针时使用了方括号，编译器很可能不会给出警告。我们的程序可能在执行过程中在没有任何警告的情况下行为异常。

智能指针和动态数组

标准库提供了一个可以管理 new 分配的数组的 unique_ptr 版本。为了用一个 unique_ptr 管理动态数组，我们必须在对象类型后面跟一对空方括号：

```
// up 指向一个包含 10 个未初始化 int 的数组
unique_ptr<int[]> up(new int[10]);
up.release(); // 自动用 delete[]销毁其指针
```

类型说明符中的方括号（<int[]>）指出 up 指向一个 int 数组而不是一个 int。由于 up 指向一个数组，当 up 销毁它管理的指针时，会自动使用 delete[]。

指向数组的 unique_ptr 提供的操作与我们在 12.1.5 节（第 417 页）中使用的那些操作有一些不同，我们在表 12.6 中描述了这些操作。当一个 unique_ptr 指向一个数组时，我们不能使用点和箭头成员运算符。毕竟 unique_ptr 指向的是一个数组而不是单个对象，因此这些运算符是无意义的。另一方面，当一个 unique_ptr 指向一个数组时，我们可以使用下标运算符来访问数组中的元素：

```
for (size_t i = 0; i != 10; ++i)
    up[i] = i; // 为每个元素赋予一个新值
```

480

表 12.6：指向数组的 unique_ptr

指向数组的 unique_ptr 不支持成员访问运算符（点和箭头运算符）。	
其他 unique_ptr 操作不变。	
unique_ptr<T[]> u	u 可以指向一个动态分配的数组，数组元素类型为 T
unique_ptr<T[]> u(p)	u 指向内置指针 p 所指向的动态分配的数组。p 必须能转换为类型 T*（参见 4.11.2 节，第 143 页）
u[i]	返回 u 拥有的数组中位置 i 处的对象
	u 必须指向一个数组

与 unique_ptr 不同，shared_ptr 不直接支持管理动态数组。如果希望使用 shared_ptr 管理一个动态数组，必须提供自己定义的删除器：

```
// 为了使用 shared_ptr，必须提供一个删除器
shared_ptr<int> sp(new int[10], [](int *p) { delete[] p; });
sp.reset(); // 使用我们提供的 lambda 释放数组，它使用 delete[]
```

本例中我们传递给 shared_ptr 一个 lambda（参见 10.3.2 节，第 346 页）作为删除器，它使用 delete[]释放数组。

如果未提供删除器，这段代码将是未定义的。默认情况下，shared_ptr 使用 delete 销毁它指向的对象。如果此对象是一个动态数组，对其使用 delete 所产生的问题与释放一个动态数组指针时忘记[]产生的问题一样（参见 12.2.1 节，第 425 页）。

shared_ptr 不直接支持动态数组管理这一特性会影响我们如何访问数组中的元素：

```
// shared_ptr 未定义下标运算符，并且不支持指针的算术运算
for (size_t i = 0; i != 10; ++i)
    *(sp.get() + i) = i; // 使用 get 获取一个内置指针
```

shared_ptr 未定义下标运算符，而且智能指针类型不支持指针算术运算。因此，为了访问数组中的元素，必须用 get 获取一个内置指针，然后用它来访问数组元素。

12.2.1 节练习

练习 12.23：编写一个程序，连接两个字符串字面常量，将结果保存在一个动态分配的 char 数组中。重写这个程序，连接两个标准库 string 对象。

练习 12.24：编写一个程序，从标准输入读取一个字符串，存入一个动态分配的字符数组中。描述你的程序如何处理变长输入。测试你的程序，输入一个超出你分配的数组长度的字符串。

练习 12.25：给定下面的 new 表达式，你应该如何释放 pa？

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

12.2.2 allocator 类

new 有一些灵活性上的局限，其中一方面表现在它将内存分配和对象构造组合在了一起。类似的，delete 将对象析构和内存释放组合在了一起。我们分配单个对象时，通常希望将内存分配和对象初始化组合在一起。因为在这种情况下，我们几乎肯定知道对象应有什么值。 481

当分配一大块内存时，我们通常计划在这块内存上按需构造对象。在此情况下，我们希望将内存分配和对象构造分离。这意味着我们可以分配大块内存，但只在真正需要时才真正执行对象创建操作（同时付出一定开销）。

一般情况下，将内存分配和对象构造组合在一起可能会导致不必要的浪费。例如：

```
string *const p = new string[n]; // 构造 n 个空 string
string s;
string *q = p;                   // q 指向第一个 string
while (cin >> s && q != p + n)
    *q++ = s;                    // 赋予*q 一个新值
const size_t size = q - p;       // 记住我们读取了多少个 string
// 使用数组
delete[] p; // p 指向一个数组；记得用 delete[] 来释放
```

new 表达式分配并初始化了 n 个 string。但是，我们可能不需要 n 个 string，少量 string 可能就足够了。这样，我们就可能创建了一些永远也用不到的对象。而且，对于那些确实要使用的对象，我们也在初始化之后立即赋予了它们新值。每个使用到的元素都被赋值了两次：第一次是在默认初始化时，随后是在赋值时。

更重要的是，那些没有默认构造函数的类就不能动态分配数组了。

allocator 类

标准库 **allocator** 类定义在头文件 memory 中，它帮助我们将内存分配和对象构造分离开来。它提供一种类型感知的内存分配方法，它分配的内存是原始的、未构造的。表 12.7 概述了 allocator 支持的操作。在本节中，我们将介绍这些 allocator 操作。在 13.5 节（第 464 页），我们将看到如何使用这个类的典型例子。

类似 vector，allocator 是一个模板（参见 3.3 节，第 86 页）。为了定义一个 allocator 对象，我们必须指明这个 allocator 可以分配的对象类型。当一个 allocator 对象分配内存时，它会根据给定的对象类型来确定恰当的内存大小和对齐位置：

```
allocator<string> alloc;              // 可以分配 string 的 allocator 对象
auto const p = alloc.allocate(n);     // 分配 n 个未初始化的 string
```

这个 allocate 调用为 n 个 string 分配了内存。

482

表 12.7：标准库 allocator 类及其算法	
allocator<T> a	定义了一个名为 a 的 allocator 对象，它可以为类型为 T 的对象分配内存
a.allocate(n)	分配一段原始的、未构造的内存，保存 n 个类型为 T 的对象
a.deallocate(p, n)	释放从 T*指针 p 中地址开始的内存，这块内存保存了 n 个类型为 T 的对象；p 必须是一个先前由 allocate 返回的指针，且 n 必须是 p 创建时所要求的大小。在调用 deallocate 之前，用户必须对每个在这块内存中创建的对象调用 destroy
a.construct(p, *args*)	p 必须是一个类型为 T*的指针，指向一块原始内存；*arg* 被传递给类型为 T 的构造函数，用来在 p 指向的内存中构造一个对象
a.destroy(p)	p 为 T*类型的指针，此算法对 p 指向的对象执行析构函数（参见 12.1.1 节，第 402 页）

allocator 分配未构造的内存

allocator 分配的内存是未构造的（unconstructed）。我们按需要在此内存中构造对象。在新标准库中，construct 成员函数接受一个指针和零个或多个额外参数，在给定位置构造一个元素。额外参数用来初始化构造的对象。类似 make_shared 的参数（参见 12.1.1 节，第 401 页），这些额外参数必须是与构造的对象的类型相匹配的合法的初始化器：

C++ 11

```
auto q = p; // q 指向最后构造的元素之后的位置
alloc.construct(q++);              // *q 为空字符串
alloc.construct(q++, 10, 'c');     // *q 为 cccccccccc
alloc.construct(q++, "hi");        // *q 为 hi!
```

在早期版本的标准库中，construct 只接受两个参数：指向创建对象位置的指针和一个元素类型的值。因此，我们只能将一个元素拷贝到未构造空间中，而不能用元素类型的任何其他构造函数来构造一个元素。

还未构造对象的情况下就使用原始内存是错误的：

```
cout << *p << endl; // 正确：使用 string 的输出运算符
cout << *q << endl; // 灾难：q 指向未构造的内存！
```

WARNING

为了使用 allocate 返回的内存，我们必须用 construct 构造对象。使用未构造的内存，其行为是未定义的。

当我们用完对象后，必须对每个构造的元素调用 destroy 来销毁它们。函数 destroy 接受一个指针，对指向的对象执行析构函数（参见 12.1.1 节，第 402 页）：

483

```
while (q != p)
    alloc.destroy(--q); // 释放我们真正构造的 string
```

在循环开始处，q 指向最后构造的元素之后的位置。我们在调用 destroy 之前对 q 进行了递减操作。因此，第一次调用 destroy 时，q 指向最后一个构造的元素。最后一步循环中我们 destroy 了第一个构造的元素，随后 q 将与 p 相等，循环结束。

WARNING

我们只能对真正构造了的元素进行 destroy 操作。

一旦元素被销毁后，就可以重新使用这部分内存来保存其他 string，也可以将其归

还给系统。释放内存通过调用 deallocate 来完成：

```
alloc.deallocate(p, n);
```

我们传递给 deallocate 的指针不能为空，它必须指向由 allocate 分配的内存。而且，传递给 deallocate 的大小参数必须与调用 allocated 分配内存时提供的大小参数具有一样的值。

拷贝和填充未初始化内存的算法

标准库还为 allocator 类定义了两个伴随算法，可以在未初始化内存中创建对象。表 12.8 描述了这些函数，它们都定义在头文件 memory 中。

表 12.8：allocator 算法	
这些函数在给定目的位置创建元素，而不是由系统分配内存给它们。	
uninitialized_copy(b,e,b2)	从迭代器 b 和 e 指出的输入范围中拷贝元素到迭代器 b2 指定的未构造的原始内存中。b2 指向的内存必须足够大，能容纳输入序列中元素的拷贝
uninitialized_copy_n(b,n,b2)	从迭代器 b 指向的元素开始，拷贝 n 个元素到 b2 开始的内存中
uninitialized_fill(b,e,t)	在迭代器 b 和 e 指定的原始内存范围中创建对象，对象的值均为 t 的拷贝
uninitialized_fill_n(b,n,t)	从迭代器 b 指向的内存地址开始创建 n 个对象。b 必须指向足够大的未构造的原始内存，能够容纳给定数量的对象

作为一个例子，假定有一个 int 的 vector，希望将其内容拷贝到动态内存中。我们将分配一块比 vector 中元素所占用空间大一倍的动态内存，然后将原 vector 中的元素拷贝到前一半空间，对后一半空间用一个给定值进行填充：

```
// 分配比 vi 中元素所占用空间大一倍的动态内存
auto p = alloc.allocate(vi.size() * 2);
// 通过拷贝 vi 中的元素来构造从 p 开始的元素
auto q = uninitialized_copy(vi.begin(), vi.end(), p);
// 将剩余元素初始化为 42
uninitialized_fill_n(q, vi.size(), 42);
```

484

类似拷贝算法（参见 10.2.2 节，第 341 页），uninitialized_copy 接受三个迭代器参数。前两个表示输入序列，第三个表示这些元素将要拷贝到的目的空间。传递给 uninitialized_copy 的目的位置迭代器必须指向未构造的内存。与 copy 不同，uninitialized_copy 在给定目的位置构造元素。

类似 copy，uninitialized_copy 返回（递增后的）目的位置迭代器。因此，一次 uninitialized_copy 调用会返回一个指针，指向最后一个构造的元素之后的位置。在本例中，我们将此指针保存在 q 中，然后将 q 传递给 uninitialized_fill_n。此函数类似 fill_n（参见 10.2.2 节，第 340 页），接受一个指向目的位置的指针、一个计数和一个值。它会在目的位置指针指向的内存中创建给定数目个对象，用给定值对它们进行初始化。

12.2.2 节练习

练习 12.26：用 allocator 重写第 427 页中的程序。

12.3 使用标准库：文本查询程序

我们将实现一个简单的文本查询程序，作为标准库相关内容学习的总结。我们的程序允许用户在一个给定文件中查询单词。查询结果是单词在文件中出现的次数及其所在行的列表。如果一个单词在一行中出现多次，此行只列出一次。行会按照升序输出——即，第 7 行会在第 9 行之前显示，依此类推。

例如，我们可能读入一个包含本章内容（指英文版中的文本）的文件，在其中寻找单词 element。输出结果的前几行应该是这样的：

```
element occurs 112 times
    (line 36) A set element contains only a key;
    (line 158) operator creates a new element
    (line 160) Regardless of whether the element
    (line 168) When we fetch an element from a map, we
    (line 214) If the element is not found, find returns
```

接下来还有大约 100 行，都是单词 element 出现的位置。

12.3.1 文本查询程序设计

485

开始一个程序的设计的一种好方法是列出程序的操作。了解需要哪些操作会帮助我们分析出需要什么样的数据结构。从需求入手，我们的文本查询程序需要完成如下任务：

- 当程序读取输入文件时，它必须记住单词出现的每一行。因此，程序需要逐行读取输入文件，并将每一行分解为独立的单词
- 当程序生成输出时，
  - 它必须能提取每个单词所关联的行号
  - 行号必须按升序出现且无重复
  - 它必须能打印给定行号中的文本。

利用多种标准库设施，我们可以很漂亮地实现这些要求：

- 我们将使用一个 vector<string>来保存整个输入文件的一份拷贝。输入文件中的每行保存为 vector 中的一个元素。当需要打印一行时，可以用行号作为下标来提取行文本。
- 我们使用一个 istringstream（参见 8.3 节，第 287 页）来将每行分解为单词。
- 我们使用一个 set 来保存每个单词在输入文本中出现的行号。这保证了每行只出现一次且行号按升序保存。
- 我们使用一个 map 来将每个单词与它出现的行号 set 关联起来。这样我们就可以方便地提取任意单词的 set。

我们的解决方案还使用了 shared_ptr，原因稍后进行解释。

数据结构

虽然我们可以用 vector、set 和 map 来直接编写文本查询程序，但如果定义一个更

为抽象的解决方案，会更为有效。我们将从定义一个保存输入文件的类开始，这会令文件查询更为容易。我们将这个类命名为 TextQuery，它包含一个 vector 和一个 map。vector 用来保存输入文件的文本，map 用来关联每个单词和它出现的行号的 set。这个类将会有一个用来读取给定输入文件的构造函数和一个执行查询的操作。

查询操作要完成的任务非常简单：查找 map 成员，检查给定单词是否出现。设计这个函数的难点是确定应该返回什么内容。一旦找到了一个单词，我们需要知道它出现了多少次、它出现的行号以及每行的文本。

返回所有这些内容的最简单的方法是定义另一个类，可以命名为 QueryResult，来保存查询结果。这个类会有一个 print 函数，完成结果打印工作。

在类之间共享数据

486

我们的 QueryResult 类要表达查询的结果。这些结果包括与给定单词关联的行号的 set 和这些行对应的文本。这些数据都保存在 TextQuery 类型的对象中。

由于 QueryResult 所需要的数据都保存在一个 TextQuery 对象中，我们就必须确定如何访问它们。我们可以拷贝行号的 set，但这样做可能很耗时。而且，我们当然不希望拷贝 vector，因为这可能会引起整个文件的拷贝，而目标只不过是为了打印文件的一小部分而已（通常会是这样）。

通过返回指向 TextQuery 对象内部的迭代器（或指针），我们可以避免拷贝操作。但是，这种方法开启了一个陷阱：如果 TextQuery 对象在对应的 QueryResult 对象之前被销毁，会发生什么？在此情况下，QueryResult 就将引用一个不再存在的对象中的数据。

对于 QueryResult 对象和对应的 TextQuery 对象的生存期应该同步这一观察结果，其实已经暗示了问题的解决方案。考虑到这两个类概念上“共享”了数据，可以使用 shared_ptr（参见 12.1.1 节，第 400 页）来反映数据结构中的这种共享关系。

使用 TextQuery 类

当我们设计一个类时，在真正实现成员之前先编写程序使用这个类，是一种非常有用的方法。通过这种方法，可以看到类是否具有我们所需要的操作。例如，下面的程序使用了 TextQuery 和 QueryResult 类。这个函数接受一个指向要处理的文件的 ifstream，并与用户交互，打印给定单词的查询结果

```
void runQueries(ifstream &infile)
{
    // infile 是一个 ifstream，指向我们要处理的文件
    TextQuery tq(infile); // 保存文件并建立查询 map
    // 与用户交互：提示用户输入要查询的单词，完成查询并打印结果
    while (true) {
        cout << "enter word to look for, or q to quit: ";
        string s;
        // 若遇到文件尾或用户输入了'q'时循环终止
        if (!(cin >> s) || s == "q") break;
        // 指向查询并打印结果
        print(cout, tq.query(s)) << endl;
    }
}
```

我们首先用给定的 ifstream 初始化一个名为 tq 的 TextQuery 对象。TextQuery 的构造函数读取输入文件，保存在 vector 中，并建立单词到所在行号的 map。

while（无限）循环提示用户输入一个要查询的单词，并打印出查询结果，如此往复。循环条件检测字面常量 true（参见 2.1.3 节，第 37 页），因此永远成功。循环的退出是通过 if 语句中的 break（参见 5.5.1 节，第 170 页）实现的。此 if 语句检查输入是否成功。如果成功，它再检查用户是否输入了 q。输入失败或用户输入了 q 都会使循环终止。一旦用户输入了要查询的单词，我们要求 tq 查找这个单词，然后调用 print 打印搜索结果。

487

12.3.1 节练习

练习 12.27：TextQuery 和 QueryResult 类只使用了我们已经介绍过的语言和标准库特性。不要提前看后续章节内容，只用已经学到的知识对这两个类编写你自己的版本。

练习 12.28：编写程序实现文本查询，不要定义类来管理数据。你的程序应该接受一个文件，并与用户交互来查询单词。使用 vector、map 和 set 容器来保存来自文件的数据并生成查询结果。

练习 12.29：我们曾经用 do while 循环来编写管理用户交互的循环（参见 5.4.4 节，第 169 页）。用 do while 重写本节程序，解释你倾向于哪个版本，为什么。

12.3.2 文本查询程序类的定义

我们以 TextQuery 类的定义开始。用户创建此类的对象时会提供一个 istream，用来读取输入文件。这个类还提供一个 query 操作，接受一个 string，返回一个 QueryResult 表示 string 出现的那些行。

设计类的数据成员时，需要考虑与 QueryResult 对象共享数据的需求。QueryResult 类需要共享保存输入文件的 vector 和保存单词关联的行号的 set。因此，这个类应该有两个数据成员：一个指向动态分配的 vector（保存输入文件）的 shared_ptr 和一个 string 到 shared_ptr<set>的 map。map 将文件中每个单词关联到一个动态分配的 set 上，而此 set 保存了该单词所出现的行号。

为了使代码更易读，我们还会定义一个类型成员（参见 7.3.1 节，第 243 页）来引用行号，即 string 的 vector 中的下标：

```
class QueryResult; // 为了定义函数 query 的返回类型，这个定义是必需的
class TextQuery {
public:
    using line_no = std::vector<std::string>::size_type;
    TextQuery(std::ifstream&);
    QueryResult query(const std::string&) const;
private:
    std::shared_ptr<std::vector<std::string>> file; // 输入文件
    // 每个单词到它所在的行号的集合的映射
    std::map<std::string,
             std::shared_ptr<std::set<line_no>>> wm;
};
```

488

这个类定义最困难的部分是解开类名。与往常一样，对于可能置于头文件中的代码，在使用标准库名字时要加上 std::（参见 3.1 节，第 74 页）。在本例中，我们反复使用了 std::，

使得代码开始可能有些难读。例如，

```
std::map<std::string, std::shared_ptr<std::set<line_no>>> wm;
```

如果写成下面的形式可能就更好理解一些

```
map<string, shared_ptr<set<line_no>>> wm;
```

TextQuery 构造函数

TextQuery 的构造函数接受一个 ifstream，逐行读取输入文件：

```
// 读取输入文件并建立单词到行号的映射
TextQuery::TextQuery(ifstream &is): file(new vector<string>)
{
    string text;
    while (getline(is, text)) {        // 对文件中每一行
        file->push_back(text);         // 保存此行文本
        int n = file->size() - 1;      // 当前行号
        istringstream line(text);      // 将行文本分解为单词
        string word;
        while (line >> word) {         // 对行中每个单词
            // 如果单词不在 wm 中，以之为下标在 wm 中添加一项
            auto &lines = wm[word];    // lines 是一个 shared_ptr
            if (!lines) // 在我们第一次遇到这个单词时，此指针为空
                lines.reset(new set<line_no>); // 分配一个新的 set
            lines->insert(n);          // 将此行号插入 set 中
        }
    }
}
```

构造函数的初始化器分配一个新的 vector 来保存输入文件中的文本。我们用 getline 逐行读取输入文件，并存入 vector 中。由于 file 是一个 shared_ptr，我们用->运算符解引用 file 来提取 file 指向的 vector 对象的 push_back 成员。

接下来我们用一个 istringstream（参见 8.3 节，第 287 页）来处理刚刚读入的一行中的每个单词。内层 while 循环用 istringstream 的输入运算符来从当前行读取每个单词，存入 word 中。在 while 循环内，我们用 map 下标运算符提取与 word 相关联的 shared_ptr<set>，并将 lines 绑定到此指针。注意，lines 是一个引用，因此改变 lines 也会改变 wm 中的元素。

若 word 不在 map 中，下标运算符会将 word 添加到 wm 中（参见 11.3.4 节，第 387 页），与 word 关联的值进行值初始化。这意味着，如果下标运算符将 word 添加到 wm 中，lines 将是一个空指针。如果 lines 为空，我们分配一个新的 set，并调用 reset 更新 lines 引用的 shared_ptr，使其指向这个新分配的 set。

不管是否创建了一个新的 set，我们都调用 insert 将当前行号添加到 set 中。由于 lines 是一个引用，对 insert 的调用会将新元素添加到 wm 中的 set 中。如果一个给定单词在同一行中出现多次，对 insert 的调用什么都不会做。 489

QueryResult 类

QueryResult 类有三个数据成员：一个 string，保存查询单词；一个 shared_ptr，指向保存输入文件的 vector；一个 shared_ptr，指向保存单词出现行号的 set。它唯

一的一个成员函数是一个构造函数，初始化这三个数据成员：

```
class QueryResult {
friend std::ostream& print(std::ostream&, const QueryResult&);
public:
QueryResult(std::string s,
            std::shared_ptr<std::set<line_no>> p,
            std::shared_ptr<std::vector<std::string>> f):
    sought(s), lines(p), file(f) { }
private:
    std::string sought; // 查询单词
    std::shared_ptr<std::set<line_no>> lines;          // 出现的行号
    std::shared_ptr<std::vector<std::string>> file;    // 输入文件
};
```

构造函数的唯一工作是将参数保存在对应的数据成员中，这是在其初始化器列表中完成的（参见 7.1.4 节，第 237 页）。

query 函数

query 函数接受一个 string 参数，即查询单词，query 用它来在 map 中定位对应的行号 set。如果找到了这个 string，query 函数构造一个 QueryResult，保存给定 string、TextQuery 的 file 成员以及从 wm 中提取的 set。

唯一的问题是：如果给定 string 未找到，我们应该返回什么？在这种情况下，没有可返回的 set。为了解决此问题，我们定义了一个局部 static 对象，它是一个指向空的行号 set 的 shared_ptr。当未找到给定单词时，我们返回此对象的一个拷贝：

```
QueryResult
TextQuery::query(const string &sought) const
{
    // 如果未找到 sought，我们将返回一个指向此 set 的指针
    static shared_ptr<set<line_no>> nodata(new set<line_no>);
    // 使用 find 而不是下标运算符来查找单词，避免将单词添加到 wm 中！
    auto loc = wm.find(sought);
    if (loc == wm.end())
        return QueryResult(sought, nodata, file); // 未找到
    else
        return QueryResult(sought, loc->second, file);
}
```

490 打印结果

print 函数在给定的流上打印出给定的 QueryResult 对象：

```
ostream &print(ostream & os, const QueryResult &qr)
{
    // 如果找到了单词，打印出现次数和所有出现的位置
    os << qr.sought << " occurs " << qr.lines->size() << " "
       << make_plural(qr.lines->size(), "time", "s") << endl;
    // 打印单词出现的每一行
    for (auto num : *qr.lines) // 对 set 中每个单词
        // 避免行号从 0 开始给用户带来的困惑
        os << "\t(line " << num + 1 << ") "
```

```
            << *(qr.file->begin() + num) << endl;
    return os;
}
```

我们调用 qr.lines 指向的 set 的 size 成员来报告单词出现了多少次。由于 set 是一个 shared_ptr，必须解引用 lines。调用 make_plural（参见 6.3.2 节，第 201 页）来根据大小是否等于 1 打印 time 或 times。

在 for 循环中，我们遍历 lines 所指向的 set。for 循环体打印行号，并按人们习惯的方式调整计数值。set 中的数值就是 vector 中元素的下标，从 0 开始编号。但大多数用户认为第一行的行号应该是 1，因此我们对每个行号都加上 1，转换为人们更习惯的形式。

我们用行号从 file 指向的 vector 中提取一行文本。回忆一下，当给一个迭代器加上一个数时，会得到 vector 中相应偏移之后位置的元素（参见 3.4.2 节，第 99 页）。因此，file->begin()+num 即为 file 指向的 vector 中第 num 个位置的元素。

注意此函数能正确处理未找到单词的情况。在此情况下，set 为空。第一条输出语句会注意到单词出现了 0 次。由于*res.lines 为空，for 循环一次也不会执行。

12.3.2 节练习

练习 12.30：定义你自己版本的 TextQuery 和 QueryResult 类，并执行 12.3.1 节（第 431 页）中的 runQueries 函数。

练习 12.31：如果用 vector 代替 set 保存行号，会有什么差别？哪种方法更好？为什么？

练习 12.32：重写 TextQuery 和 QueryResult 类，用 StrBlob 代替 vector<string> 保存输入文件。

练习 12.33：在第 15 章中我们将扩展查询系统，在 QueryResult 类中将会需要一些额外的成员。添加名为 begin 和 end 的成员，返回一个迭代器，指向一个给定查询返回的行号的 set 中的位置。再添加一个名为 get_file 的成员，返回一个 shared_ptr，指向 QueryResult 对象中的文件。

491 小结

在 C++中，内存是通过 new 表达式分配，通过 delete 表达式释放的。标准库还定义了一个 allocator 类来分配动态内存块。

分配动态内存的程序应负责释放它所分配的内存。内存的正确释放是非常容易出错的地方：要么内存永远不会被释放，要么在仍有指针引用它时就被释放了。新的标准库定义了智能指针类型——shared_ptr、unique_ptr 和 weak_ptr，可令动态内存管理更为安全。对于一块内存，当没有任何用户使用它时，智能指针会自动释放它。现代 C++程序应尽可能使用智能指针。

术语表

allocator 标准库类，用来分配未构造的内存。

空悬指针（dangling pointer） 一个指针，指向曾经保存一个对象但现在已释放的内存。众所周知，空悬指针引起的程序错误非常难以调试。

delete 释放 new 分配的内存。delete p 释放对象，delete []p 释放 p 指向的数组。p 可以为空，或者指向 new 分配的内存。

释放器（deleter） 传递给智能指针的函数，用来代替 delete 释放指针绑定的对象。

析构函数（destructor） 特殊的成员函数，负责在对象离开作用域或被释放时完成清理工作。

动态分配的（dynamically allocated） 在自由空间中分配的对象。在自由空间中分配的对象直到被显式释放或程序结束才会销毁。

自由空间（free store） 程序可用的内存池，保存动态分配的对象。

堆（heap） 自由空间的同义词。

new 从自由空间分配内存。new T 分配并构造一个类型为 T 的对象，并返回一个指向该对象的指针。如果 T 是一个数组类型，new 返回一个指向数组首元素的指针。类似的，new [n] T 分配 n 个类型为 T 的对象，并返回指向数组首元素的指针。默认情况下，分配的对象进行默认初始化。我们也可以提供可选的初始化器。

定位 new（placement new） 一种 new 表达式形式，接受一些额外的参数，在 new 关键字后面的括号中给出。例如，new (nothrow) int 告诉 new 不要抛出异常。

引用计数（reference count） 一个计数器，记录有多少用户共享一个对象。智能指针用它来判断什么时候释放所指向的对象是安全的。

shared_ptr 提供所有权共享的智能指针：对共享对象来说，当最后一个指向它的 shared_ptr 被销毁时会被释放。

智能指针（smart pointer） 标准库类型，行为类似指针，但可以检查什么时候使用指针是安全的。智能指针类型负责在恰当的时候释放内存。

unique_ptr 提供独享所有权的智能指针：当 unique_ptr 被销毁时，它指向的对象被释放。unique_ptr 不能直接拷贝或赋值。

weak_ptr 一种智能指针，指向由 shared_ptr 管理的对象。在确定是否应释放对象时，shared_ptr 并不把 weak_ptr 统计在内。

第Ⅲ部分
类设计者的工具

内容

第 13 章　拷贝控制 439
第 14 章　操作重载与类型转换 489
第 15 章　面向对象程序设计 525
第 16 章　模板与泛型编程 577

类是 C++的核心概念。我们已经从第 7 章开始详细介绍了如何定义类。第 7 章涵盖了使用类的所有基本知识：类作用域、数据隐藏以及构造函数，还介绍了类的一些重要特性：成员函数、隐式 this 指针、友元以及 const、static 和 mutable 成员。在第Ⅲ部分中，我们将延伸类的有关话题的讨论，将介绍拷贝控制、重载运算符、继承和模板。

如前所述，在 C++中，我们通过定义构造函数来控制在类类型的对象初始化时做什么。类还可以控制在对象拷贝、赋值、移动和销毁时做什么。在这方面，C++与其他语言是不同的，其他很多语言都没有给予类设计者控制这些操作的能力。第 13 章将介绍这些内容。本章还会介绍新标准引入的两个重要概念：右值引用和移动操作。

第 14 章介绍运算符重载，这种机制允许内置运算符作用于类类型的运算对象。这样，我们创建的类型直观上就可以像内置类型一样使用，运算符重载是 C++借以实现这一目的的方法之一。

类可以重载的运算符中有一种特殊的运算符——函数调用运算符。对于重载了这种运算符的类，我们可以“调用”其对象，就好像它们是函数一样。新标准库中提供了一些设施，使得不同类型的可调用对象可以以一种一致的方式来使用，我们也将介绍这部分内容。

第 14 章最后将介绍另一种特殊类型的类成员函数——转换运算符。这些运算符定义了类类型对象的隐式转换机制。编译器应用这种转换机制的场合与原因都与内置类型转换是一样的。

第Ⅲ部分的最后两章将介绍 C++如何支持面向对象编程和泛型编程。

第 15 章介绍继承和动态绑定。继承和动态绑定与数据抽象一起构成了面向对象编程的基础。继承令关联类型的定义更为简单，而动态绑定可以帮助我们编写类型无关的代码，可以忽略具有继承关系的类型之间的差异。

第 16 章介绍函数模板和类模板。模板可以让我们写出类型无关的通用类和函数。新标准引入了一些模板相关的新特性：可变参数模板、模板类型别名以及控制实例化的新方法。

编写我们自己的面向对象的或是泛型的类型需要对 C++有深刻的理解。幸运的是，我们无须掌握如何构建面向对象和泛型类型的细节也可以使用它们。例如，标准库中广泛使用了我们将在第 15 章和第 16 章中学习的技术，虽然我们已经使用过标准库类型和算法，但实际上我们并不了解它们是如何实现的。

因此，读者应该明白第III部分涉及的是相当深入的内容。编写模板或面向对象的类要求对 C++的基本知识和基本类的定义有着深刻的理解。

第 13 章
拷贝控制

内容

13.1 拷贝、赋值与销毁 440
13.2 拷贝控制和资源管理 452
13.3 交换操作 457
13.4 拷贝控制示例 460
13.5 动态内存管理类 464
13.6 对象移动 470
小结 486
术语表 486

如我们在第 7 章所见，每个类都定义了一个新类型和在此类型对象上可执行的操作。在本章中，我们还将学到，类可以定义构造函数，用来控制在创建此类型对象时做什么。

在本章中，我们还将学习类如何控制该类型对象拷贝、赋值、移动或销毁时做什么。类通过一些特殊的成员函数控制这些操作，包括：拷贝构造函数、移动构造函数、拷贝赋值运算符、移动赋值运算符以及析构函数。

496 当定义一个类时，我们显式地或隐式地指定在此类型的对象拷贝、移动、赋值和销毁时做什么。一个类通过定义五种特殊的成员函数来控制这些操作，包括：**拷贝构造函数**（copy constructor）、**拷贝赋值运算符**（copy-assignment operator）、**移动构造函数**（move constructor）、**移动赋值运算符**（move-assignment operator）和**析构函数**（destructor）。拷贝和移动构造函数定义了当用同类型的另一个对象初始化本对象时做什么。拷贝和移动赋值运算符定义了将一个对象赋予同类型的另一个对象时做什么。析构函数定义了当此类型对象销毁时做什么。我们称这些操作为**拷贝控制操作**（copy control）。

如果一个类没有定义所有这些拷贝控制成员，编译器会自动为它定义缺失的操作。因此，很多类会忽略这些拷贝控制操作（参见 7.1.5 节，第 239 页）。但是，对一些类来说，依赖这些操作的默认定义会导致灾难。通常，实现拷贝控制操作最困难的地方是首先认识到什么时候需要定义这些操作。

在定义任何 C++类时，拷贝控制操作都是必要部分。对初学 C++的程序员来说，必须定义对象拷贝、移动、赋值或销毁时做什么，这常常令他们感到困惑。这种困扰很复杂，因为如果我们不显式定义这些操作，编译器也会为我们定义，但编译器定义的版本的行为可能并非我们所想。

13.1 拷贝、赋值与销毁

我们将以最基本的操作——拷贝构造函数、拷贝赋值运算符和析构函数作为开始。我们在 13.6 节（第 470 页）中将介绍移动操作（新标准所引入的操作）。

13.1.1 拷贝构造函数

如果一个构造函数的第一个参数是自身类类型的引用，且任何额外参数都有默认值，则此构造函数是拷贝构造函数。

```
class Foo {
public:
    Foo();              // 默认构造函数
    Foo(const Foo&);    // 拷贝构造函数
    // ...
};
```

拷贝构造函数的第一个参数必须是一个引用类型，原因我们稍后解释。虽然我们可以定义一个接受非 const 引用的拷贝构造函数，但此参数几乎总是一个 const 的引用。拷贝构造函数在几种情况下都会被隐式地使用。因此，拷贝构造函数通常不应该是 explicit 的（参见 7.5.4 节，第 265 页）。

497 **合成拷贝构造函数**

如果我们没有为一个类定义拷贝构造函数，编译器会为我们定义一个。与合成默认构造函数（参见 7.1.4 节，第 235 页）不同，即使我们定义了其他构造函数，编译器也会为我们合成一个拷贝构造函数。

如我们将在 13.1.6 节（第 450 页）中所见，对某些类来说，**合成拷贝构造函数**（synthesized copy constructor）用来阻止我们拷贝该类类型的对象。而一般情况，合成的拷贝构造函数会将其参数的成员逐个拷贝到正在创建的对象中（参见 7.1.5 节，第 239 页）。编译器从给

定对象中依次将每个非 static 成员拷贝到正在创建的对象中。

每个成员的类型决定了它如何拷贝：对类类型的成员，会使用其拷贝构造函数来拷贝；内置类型的成员则直接拷贝。虽然我们不能直接拷贝一个数组（参见 3.5.1 节，第 102 页），但合成拷贝构造函数会逐元素地拷贝一个数组类型的成员。如果数组元素是类类型，则使用元素的拷贝构造函数来进行拷贝。

作为一个例子，我们的 Sales_data 类的合成拷贝构造函数等价于：

```
class Sales_data {
public:
    // 其他成员和构造函数的定义，如前
    // 与合成的拷贝构造函数等价的拷贝构造函数的声明
   Sales_data(const Sales_data&);
private:
    std::string bookNo;
    int units_sold = 0;
    double revenue = 0.0;
};
// 与 Sales_data 的合成的拷贝构造函数等价
Sales_data::Sales_data(const Sales_data &orig):
    bookNo(orig.bookNo),          // 使用 string 的拷贝构造函数
    units_sold(orig.units_sold),  // 拷贝 orig.units_sold
    revenue(orig.revenue)         // 拷贝 orig.revenue
    {    }                        // 空函数体
```

拷贝初始化

现在，我们可以完全理解直接初始化和拷贝初始化之间的差异了（参见 3.2.1 节，第 76 页）：

```
string dots(10, '.');                  // 直接初始化
string s(dots);                        // 直接初始化
string s2 = dots;                      // 拷贝初始化
string null_book = "9-999-99999-9";    // 拷贝初始化
string nines = string(100, '9');       // 拷贝初始化
```

当使用直接初始化时，我们实际上是要求编译器使用普通的函数匹配（参见 6.4 节，第 209 页）来选择与我们提供的参数最匹配的构造函数。当我们使用**拷贝初始化**（copy initialization）时，我们要求编译器将右侧运算对象拷贝到正在创建的对象中，如果需要的话还要进行类型转换（参见 7.5.4 节，第 263 页）。

拷贝初始化通常使用拷贝构造函数来完成。但是，如我们将在 13.6.2 节（第 473 页）所见，如果一个类有一个移动构造函数，则拷贝初始化有时会使用移动构造函数而非拷贝构造函数来完成。但现在，我们只需了解拷贝初始化何时发生，以及拷贝初始化是依靠拷贝构造函数或移动构造函数来完成的就可以了。 498

拷贝初始化不仅在我们用=定义变量时会发生，在下列情况下也会发生

- 将一个对象作为实参传递给一个非引用类型的形参
- 从一个返回类型为非引用类型的函数返回一个对象
- 用花括号列表初始化一个数组中的元素或一个聚合类中的成员（参见 7.5.5 节，第 266 页）

某些类类型还会对它们所分配的对象使用拷贝初始化。例如，当我们初始化标准库容器或是调用其 insert 或 push 成员（参见 9.3.1 节，第 306 页）时，容器会对其元素进行拷贝初始化。与之相对，用 emplace 成员创建的元素都进行直接初始化（参见 9.3.1 节，第 308 页）。

参数和返回值

在函数调用过程中，具有非引用类型的参数要进行拷贝初始化（参见 6.2.1 节，第 188 页）。类似的，当一个函数具有非引用的返回类型时，返回值会被用来初始化调用方的结果（参见 6.3.2 节，第 201 页）。

拷贝构造函数被用来初始化非引用类类型参数，这一特性解释了为什么拷贝构造函数自己的参数必须是引用类型。如果其参数不是引用类型，则调用永远也不会成功——为了调用拷贝构造函数，我们必须拷贝它的实参，但为了拷贝实参，我们又需要调用拷贝构造函数，如此无限循环。

拷贝初始化的限制

如前所述，如果我们使用的初始化值要求通过一个 explicit 的构造函数来进行类型转换（参见 7.5.4 节，第 265 页），那么使用拷贝初始化还是直接初始化就不是无关紧要的了：

```
vector<int> v1(10);  // 正确：直接初始化
vector<int> v2 = 10; // 错误：接受大小参数的构造函数是 explicit 的
void f(vector<int>); // f 的参数进行拷贝初始化
f(10); // 错误：不能用一个 explicit 的构造函数拷贝一个实参
f(vector<int>(10));  // 正确：从一个 int 直接构造一个临时 vector
```

直接初始化 v1 是合法的，但看起来与之等价的拷贝初始化 v2 则是错误的，因为 vector 的接受单一大小参数的构造函数是 explicit 的。出于同样的原因，当传递一个实参或从函数返回一个值时，我们不能隐式使用一个 explicit 构造函数。如果我们希望使用一个 explicit 构造函数，就必须显式地使用，像此代码中最后一行那样。

499 >

编译器可以绕过拷贝构造函数

在拷贝初始化过程中，编译器可以（但不是必须）跳过拷贝/移动构造函数，直接创建对象。即，编译器被允许将下面的代码

```
string null_book = "9-999-99999-9"; // 拷贝初始化
```

改写为

```
string null_book("9-999-99999-9"); // 编译器略过了拷贝构造函数
```

但是，即使编译器略过了拷贝/移动构造函数，但在这个程序点上，拷贝/移动构造函数必须是存在且可访问的（例如，不能是 private 的）。

13.1.1 节练习

练习 13.1：拷贝构造函数是什么？什么时候使用它？

练习 13.2：解释为什么下面的声明是非法的：

```
Sales_data::Sales_data(Sales_data rhs);
```

练习 13.3：当我们拷贝一个 StrBlob 时，会发生什么？拷贝一个 StrBlobPtr 呢？

练习 13.4：假定 Point 是一个类类型，它有一个 public 的拷贝构造函数，指出下面程序片段中哪些地方使用了拷贝构造函数：

```
Point global;
Point foo_bar(Point arg)
{
    Point local = arg, *heap = new Point(global);
    *heap = local;
    Point pa[ 4 ] = { local, *heap };
    return *heap;
}
```

练习 13.5：给定下面的类框架，编写一个拷贝构造函数，拷贝所有成员。你的构造函数应该动态分配一个新的 string（参见 12.1.2 节，第 407 页），并将对象拷贝到 ps 指向的位置，而不是 ps 本身的位置。

```
class HasPtr {
public:
    HasPtr(const std::string &s = std::string()):
        ps(new std::string(s)), i(0) { }
private:
    std::string *ps;
    int i;
};
```

13.1.2 拷贝赋值运算符

500

与类控制其对象如何初始化一样，类也可以控制其对象如何赋值：

```
Sales_data trans, accum;
trans = accum; // 使用 Sales_data 的拷贝赋值运算符
```

与拷贝构造函数一样，如果类未定义自己的拷贝赋值运算符，编译器会为它合成一个。

重载赋值运算符

在介绍合成赋值运算符之前，我们需要了解一点儿有关**重载运算符**（overloaded operator）的知识，详细内容将在第 14 章中进行介绍。

重载运算符本质上是函数，其名字由 operator 关键字后接表示要定义的运算符的符号组成。因此，赋值运算符就是一个名为 operator=的函数。类似于任何其他函数，运算符函数也有一个返回类型和一个参数列表。

重载运算符的参数表示运算符的运算对象。某些运算符，包括赋值运算符，必须定义为成员函数。如果一个运算符是一个成员函数，其左侧运算对象就绑定到隐式的 this 参数（参见 7.1.2 节，第 231 页）。对于一个二元运算符，例如赋值运算符，其右侧运算对象作为显式参数传递。

拷贝赋值运算符接受一个与其所在类相同类型的参数：

```
class Foo {
public:
```

```
    Foo& operator=(const Foo&); // 赋值运算符
    // ...
};
```

为了与内置类型的赋值（参见 4.4 节，第 129 页）保持一致，赋值运算符通常返回一个指向其左侧运算对象的引用。另外值得注意的是，标准库通常要求保存在容器中的类型要具有赋值运算符，且其返回值是左侧运算对象的引用。

Best Practices 赋值运算符通常应该返回一个指向其左侧运算对象的引用。

合成拷贝赋值运算符

与处理拷贝构造函数一样，如果一个类未定义自己的拷贝赋值运算符，编译器会为它生成一个**合成拷贝赋值运算符**（synthesized copy-assignment operator）。类似拷贝构造函数，对于某些类，合成拷贝赋值运算符用来禁止该类型对象的赋值（参见 13.1.6 节，第 450 页）。如果拷贝赋值运算符并非出于此目的，它会将右侧运算对象的每个非 static 成员赋予左侧运算对象的对应成员，这一工作是通过成员类型的拷贝赋值运算符来完成的。对于数组类型的成员，逐个赋值数组元素。合成拷贝赋值运算符返回一个指向其左侧运算对象的引用。

501 作为一个例子，下面的代码等价于 Sales_data 的合成拷贝赋值运算符：

```
// 等价于合成拷贝赋值运算符
Sales_data&
Sales_data::operator=(const Sales_data &rhs)
{
    bookNo = rhs.bookNo;          // 调用 string::operator=
    units_sold = rhs.units_sold;  // 使用内置的 int 赋值
    revenue = rhs.revenue;        // 使用内置的 double 赋值
    return *this;                 // 返回一个此对象的引用
}
```

13.1.2 节练习

练习 13.6：拷贝赋值运算符是什么？什么时候使用它？合成拷贝赋值运算符完成什么工作？什么时候会生成合成拷贝赋值运算符？

练习 13.7：当我们将一个 StrBlob 赋值给另一个 StrBlob 时，会发生什么？赋值 StrBlobPtr 呢？

练习 13.8：为 13.1.1 节（第 443 页）练习 13.5 中的 HasPtr 类编写赋值运算符。类似拷贝构造函数，你的赋值运算符应该将对象拷贝到 ps 指向的位置。

13.1.3 析构函数

析构函数执行与构造函数相反的操作：构造函数初始化对象的非 static 数据成员，还可能做一些其他工作；析构函数释放对象使用的资源，并销毁对象的非 static 数据成员。

析构函数是类的一个成员函数，名字由波浪号接类名构成。它没有返回值，也不接受参数：

```
class Foo {
public:
    ~Foo(); // 析构函数
```

```
    //...
};
```

由于析构函数不接受参数，因此它不能被重载。对一个给定类，只会有唯一一个析构函数。

析构函数完成什么工作

如同构造函数有一个初始化部分和一个函数体（参见 7.5.1 节，第 257 页），析构函数也有一个函数体和一个析构部分。在一个构造函数中，成员的初始化是在函数体执行之前完成的，且按照它们在类中出现的顺序进行初始化。在一个析构函数中，首先执行函数体，然后销毁成员。成员按初始化顺序的逆序销毁。 502

在对象最后一次使用之后，析构函数的函数体可执行类设计者希望执行的任何收尾工作。通常，析构函数释放对象在生存期分配的所有资源。

在一个析构函数中，不存在类似构造函数中初始化列表的东西来控制成员如何销毁，析构部分是隐式的。成员销毁时发生什么完全依赖于成员的类型。销毁类类型的成员需要执行成员自己的析构函数。内置类型没有析构函数，因此销毁内置类型成员什么也不需要做。

隐式销毁一个内置指针类型的成员不会 delete 它所指向的对象。

与普通指针不同，智能指针（参见 12.1.1 节，第 402 页）是类类型，所以具有析构函数。因此，与普通指针不同，智能指针成员在析构阶段会被自动销毁。

什么时候会调用析构函数

无论何时一个对象被销毁，就会自动调用其析构函数：

- 变量在离开其作用域时被销毁。
- 当一个对象被销毁时，其成员被销毁。
- 容器（无论是标准库容器还是数组）被销毁时，其元素被销毁。
- 对于动态分配的对象，当对指向它的指针应用 delete 运算符时被销毁（参见 12.1.2 节，第 409 页）。
- 对于临时对象，当创建它的完整表达式结束时被销毁。

由于析构函数自动运行，我们的程序可以按需要分配资源，而（通常）无须担心何时释放这些资源。

例如，下面代码片段定义了四个 Sales_data 对象：

```
{ // 新作用域
    // p 和 p2 指向动态分配的对象
    Sales_data *p = new Sales_data;        // p 是一个内置指针
    auto p2 = make_shared<Sales_data>();  // p2 是一个 shared_ptr
    Sales_data item(*p);       // 拷贝构造函数将*p 拷贝到 item 中
    vector<Sales_data> vec;    // 局部对象
    vec.push_back(*p2);        // 拷贝 p2 指向的对象
    delete p;                  // 对 p 指向的对象执行析构函数
} // 退出局部作用域；对 item、p2 和 vec 调用析构函数
  // 销毁 p2 会递减其引用计数；如果引用计数变为 0，对象被释放
  // 销毁 vec 会销毁它的元素
```

503 每个 Sales_data 对象都包含一个 string 成员，它分配动态内存来保存 bookNo 成员中的字符。但是，我们的代码唯一需要直接管理的内存就是我们直接分配的 Sales_data 对象。我们的代码只需直接释放绑定到 p 的动态分配对象。

其他 Sales_data 对象会在离开作用域时被自动销毁。当程序块结束时，vec、p2 和 item 都离开了作用域，意味着在这些对象上分别会执行 vector、shared_ptr 和 Sales_data 的析构函数。vector 的析构函数会销毁我们添加到 vec 的元素。shared_ptr 的析构函数会递减 p2 指向的对象的引用计数。在本例中，引用计数会变为 0，因此 shared_ptr 的析构函数会 delete p2 分配的 Sales_data 对象。

在所有情况下，Sales_data 的析构函数都会隐式地销毁 bookNo 成员。销毁 bookNo 会调用 string 的析构函数，它会释放用来保存 ISBN 的内存。

当指向一个对象的引用或指针离开作用域时，析构函数不会执行。

合成析构函数

当一个类未定义自己的析构函数时，编译器会为它定义一个**合成析构函数**(synthesized destructor)。类似拷贝构造函数和拷贝赋值运算符，对于某些类，合成析构函数被用来阻止该类型的对象被销毁（参见 13.1.6 节，第 450 页）。如果不是这种情况，合成析构函数的函数体就为空。

例如，下面的代码片段等价于 Sales_data 的合成析构函数：

```
class Sales_data {
public:
    // 成员会被自动销毁，除此之外不需要做其他事情
     ~Sales_data() { }
    // 其他成员的定义，如前
};
```

在（空）析构函数体执行完毕后，成员会被自动销毁。特别的，string 的析构函数会被调用，它将释放 bookNo 成员所用的内存。

认识到析构函数体自身并不直接销毁成员是非常重要的。成员是在析构函数体之后隐含的析构阶段中被销毁的。在整个对象销毁过程中，析构函数体是作为成员销毁步骤之外的另一部分而进行的。

13.1.3 节练习

练习 13.9：析构函数是什么？合成析构函数完成什么工作？什么时候会生成合成析构函数？

练习 13.10：当一个 StrBlob 对象销毁时会发生什么？一个 StrBlobPtr 对象销毁时呢？

练习 13.11：为前面练习中的 HasPtr 类添加一个析构函数。

练习 13.12：在下面的代码片段中会发生几次析构函数调用？

```
bool fcn(const Sales_data *trans, Sales_data accum)
{
    Sales_data item1(*trans), item2(accum);
```

```
        return item1.isbn() != item2.isbn();
    }
```

练习 13.13：理解拷贝控制成员和构造函数的一个好方法是定义一个简单的类，为该类定义这些成员，每个成员都打印出自己的名字：

```
struct X {
    X() {std::cout << "X()" << std::endl;}
    X(const X&) {std::cout << "X(const X&)" << std::endl;}
};
```

给 X 添加拷贝赋值运算符和析构函数，并编写一个程序以不同方式使用 X 的对象：将它们作为非引用和引用参数传递；动态分配它们；将它们存放于容器中；诸如此类。观察程序的输出，直到你确认理解了什么时候会使用拷贝控制成员，以及为什么会使用它们。当你观察程序输出时，记住编译器可以略过对拷贝构造函数的调用。

13.1.4 三/五法则

如前所述，有三个基本操作可以控制类的拷贝操作：拷贝构造函数、拷贝赋值运算符和析构函数。而且，在新标准下，一个类还可以定义一个移动构造函数和一个移动赋值运算符，我们将在 13.6 节（第 470 页）中介绍这些内容。

C++语言并不要求我们定义所有这些操作：可以只定义其中一个或两个，而不必定义所有。但是，这些操作通常应该被看作一个整体。通常，只需要其中一个操作，而不需要定义所有操作的情况是很少见的。 504

需要析构函数的类也需要拷贝和赋值操作

当我们决定一个类是否要定义它自己版本的拷贝控制成员时，一个基本原则是首先确定这个类是否需要一个析构函数。通常，对析构函数的需求要比对拷贝构造函数或赋值运算符的需求更为明显。如果这个类需要一个析构函数，我们几乎可以肯定它也需要一个拷贝构造函数和一个拷贝赋值运算符。

我们在练习中用过的 HasPtr 类是一个好例子（参见 13.1.1 节，第 443 页）。这个类在构造函数中分配动态内存。合成析构函数不会 delete 一个指针数据成员。因此，此类需要定义一个析构函数来释放构造函数分配的内存。

应该怎么做可能还有点儿不清晰，但基本原则告诉我们，HasPtr 也需要一个拷贝构造函数和一个拷贝赋值运算符。

如果我们为 HasPtr 定义一个析构函数，但使用合成版本的拷贝构造函数和拷贝赋值运算符，考虑会发生什么： 505

```
class HasPtr {
public:
    HasPtr(const std::string &s = std::string()):
        ps(new std::string(s)), i(0) { }
    ~HasPtr() { delete ps; }
    // 错误：HasPtr 需要一个拷贝构造函数和一个拷贝赋值运算符
    // 其他成员的定义，如前
};
```

在这个版本的类定义中，构造函数中分配的内存将在 HasPtr 对象销毁时被释放。但不幸的是，我们引入了一个严重的错误！这个版本的类使用了合成的拷贝构造函数和拷贝

赋值运算符。这些函数简单拷贝指针成员，这意味着多个 HasPtr 对象可能指向相同的内存：

```
HasPtr f(HasPtr hp)        // HasPtr 是传值参数，所以将被拷贝
{
    HasPtr ret = hp;       // 拷贝给定的 HasPtr
    // 处理 ret
    return ret;            // ret 和 hp 被销毁
}
```

当 f 返回时，hp 和 ret 都被销毁，在两个对象上都会调用 HasPtr 的析构函数。此析构函数会 delete ret 和 hp 中的指针成员。但这两个对象包含相同的指针值。此代码会导致此指针被 delete 两次，这显然是一个错误（参见 12.1.2 节，第 411 页）。将要发生什么是未定义的。

此外，f 的调用者还会使用传递给 f 的对象：

```
HasPtr p("some values");
f(p);              // 当 f 结束时，p.ps 指向的内存被释放
HasPtr q(p);       // 现在 p 和 q 都指向无效内存！
```

p（以及 q）指向的内存不再有效，在 hp（或 ret！）销毁时它就被归还给系统了。

如果一个类需要自定义析构函数，几乎可以肯定它也需要自定义拷贝赋值运算符和拷贝构造函数。

需要拷贝操作的类也需要赋值操作，反之亦然

虽然很多类需要定义所有（或是不需要定义任何）拷贝控制成员，但某些类所要完成的工作，只需要拷贝或赋值操作，不需要析构函数。

作为一个例子，考虑一个类为每个对象分配一个独有的、唯一的序号。这个类需要一个拷贝构造函数为每个新创建的对象生成一个新的、独一无二的序号。除此之外，这个拷贝构造函数从给定对象拷贝所有其他数据成员。这个类还需要自定义拷贝赋值运算符来避免将序号赋予目的对象。但是，这个类不需要自定义析构函数。

506

这个例子引出了第二个基本原则：如果一个类需要一个拷贝构造函数，几乎可以肯定它也需要一个拷贝赋值运算符。反之亦然——如果一个类需要一个拷贝赋值运算符，几乎可以肯定它也需要一个拷贝构造函数。然而，无论是需要拷贝构造函数还是需要拷贝赋值运算符都不必然意味着也需要析构函数。

13.1.4 节练习

练习 13.14：假定 numbered 是一个类，它有一个默认构造函数，能为每个对象生成一个唯一的序号，保存在名为 mysn 的数据成员中。假定 numbered 使用合成的拷贝控制成员，并给定如下函数：

```
void f (numbered s) { cout << s.mysn << endl; }
```

则下面代码输出什么内容？

```
numbered a, b = a, c = b;
f(a); f(b); f(c);
```

练习 13.15：假定 numbered 定义了一个拷贝构造函数，能生成一个新的序号。这会改变上一题中调用的输出结果吗？如果会改变，为什么？新的输出结果是什么？

练习 13.16：如果 f 中的参数是 const numbered&，将会怎样？这会改变输出结果吗？如果会改变，为什么？新的输出结果是什么？

练习 13.17：分别编写前三题中所描述的 numbered 和 f，验证你是否正确预测了输出结果。

13.1.5 使用=default

我们可以通过将拷贝控制成员定义为=default 来显式地要求编译器生成合成的版本（参见 7.1.4 节，第 237 页）：

C++ 11

```
class Sales_data {
public:
    // 拷贝控制成员；使用 default
    Sales_data() = default;
    Sales_data(const Sales_data&) = default;
    Sales_data& operator=(const Sales_data &);
    ~Sales_data() = default;
    // 其他成员的定义，如前
};
Sales_data& Sales_data::operator=(const Sales_data&) = default;
```

当我们在类内用=default 修饰成员的声明时，合成的函数将隐式地声明为内联的（就像任何其他类内声明的成员函数一样）。如果我们不希望合成的成员是内联函数，应该只对成员的类外定义使用=default，就像对拷贝赋值运算符所做的那样。 507

我们只能对具有合成版本的成员函数使用=default（即，默认构造函数或拷贝控制成员）。

13.1.6 阻止拷贝

大多数类应该定义默认构造函数、拷贝构造函数和拷贝赋值运算符，无论是隐式地还是显式地。

虽然大多数类应该定义（而且通常也的确定义了）拷贝构造函数和拷贝赋值运算符，但对某些类来说，这些操作没有合理的意义。在此情况下，定义类时必须采用某种机制阻止拷贝或赋值。例如，iostream 类阻止了拷贝，以避免多个对象写入或读取相同的 IO 缓冲。为了阻止拷贝，看起来可能应该不定义拷贝控制成员。但是，这种策略是无效的：如果我们的类未定义这些操作，编译器为它生成合成的版本。

定义删除的函数

在新标准下，我们可以通过将拷贝构造函数和拷贝赋值运算符定义为**删除的函数**（deleted function）来阻止拷贝。删除的函数是这样一种函数：我们虽然声明了它们，但不能以任何方式使用它们。在函数的参数列表后面加上=delete 来指出我们希望将它定义为删除的：

C++ 11

```
struct NoCopy {
    NoCopy() = default;        // 使用合成的默认构造函数
    NoCopy(const NoCopy&) = delete;                // 阻止拷贝
    NoCopy &operator=(const NoCopy&) = delete;     // 阻止赋值
```

```
    ~NoCopy() = default;        // 使用合成的析构函数
    // 其他成员
};
```

=delete 通知编译器（以及我们代码的读者），我们不希望定义这些成员。

与=default 不同，=delete 必须出现在函数第一次声明的时候，这个差异与这些声明的含义在逻辑上是吻合的。一个默认的成员只影响为这个成员而生成的代码，因此=default 直到编译器生成代码时才需要。而另一方面，编译器需要知道一个函数是删除的，以便禁止试图使用它的操作。

与=default 的另一个不同之处是，我们可以对任何函数指定=delete（我们只能对编译器可以合成的默认构造函数或拷贝控制成员使用=default）。虽然删除函数的主要用途是禁止拷贝控制成员，但当我们希望引导函数匹配过程时，删除函数有时也是有用的。

508

析构函数不能是删除的成员

值得注意的是，我们不能删除析构函数。如果析构函数被删除，就无法销毁此类型的对象了。对于一个删除了析构函数的类型，编译器将不允许定义该类型的变量或创建该类的临时对象。而且，如果一个类有某个成员的类型删除了析构函数，我们也不能定义该类的变量或临时对象。因为如果一个成员的析构函数是删除的，则该成员无法被销毁。而如果一个成员无法被销毁，则对象整体也就无法被销毁了。

对于删除了析构函数的类型，虽然我们不能定义这种类型的变量或成员，但可以动态分配这种类型的对象。但是，不能释放这些对象：

```
struct NoDtor {
    NoDtor() = default;  // 使用合成默认构造函数
    ~NoDtor() = delete;  // 我们不能销毁 NoDtor 类型的对象
};
NoDtor nd; // 错误：NoDtor 的析构函数是删除的
NoDtor *p = new NoDtor();// 正确：但我们不能 delete p
delete p; // 错误：NoDtor 的析构函数是删除的
```

对于析构函数已删除的类型，不能定义该类型的变量或释放指向该类型动态分配对象的指针。

合成的拷贝控制成员可能是删除的

如前所述，如果我们未定义拷贝控制成员，编译器会为我们定义合成的版本。类似的，如果一个类未定义构造函数，编译器会为其合成一个默认构造函数（参见 7.1.4 节，第 235 页）。对某些类来说，编译器将这些合成的成员定义为删除的函数：

- 如果类的某个成员的析构函数是删除的或不可访问的（例如，是 private 的），则类的合成析构函数被定义为删除的。
- 如果类的某个成员的拷贝构造函数是删除的或不可访问的，则类的合成拷贝构造函数被定义为删除的。如果类的某个成员的析构函数是删除的或不可访问的，则类合成的拷贝构造函数也被定义为删除的。
- 如果类的某个成员的拷贝赋值运算符是删除的或不可访问的，或是类有一个 const 的或引用成员，则类的合成拷贝赋值运算符被定义为删除的。
- 如果类的某个成员的析构函数是删除的或不可访问的，或是类有一个引用成员，它没有类内初始化器（参见 2.6.1 节，第 65 页），或是类有一个 const 成员，它没有

类内初始化器且其类型未显式定义默认构造函数，则该类的默认构造函数被定义为删除的。

本质上，这些规则的含义是：如果一个类有数据成员不能默认构造、拷贝、复制或销毁，则对应的成员函数将被定义为删除的。 509

一个成员有删除的或不可访问的析构函数会导致合成的默认和拷贝构造函数被定义为删除的，这看起来可能有些奇怪。其原因是，如果没有这条规则，我们可能会创建出无法销毁的对象。

对于具有引用成员或无法默认构造的 const 成员的类，编译器不会为其合成默认构造函数，这应该不奇怪。同样不出人意料的规则是：如果一个类有 const 成员，则它不能使用合成的拷贝赋值运算符。毕竟，此运算符试图赋值所有成员，而将一个新值赋予一个 const 对象是不可能的。

虽然我们可以将一个新值赋予一个引用成员，但这样做改变的是引用指向的对象的值，而不是引用本身。如果为这样的类合成拷贝赋值运算符，则赋值后，左侧运算对象仍然指向与赋值前一样的对象，而不会与右侧运算对象指向相同的对象。由于这种行为看起来并不是我们所期望的，因此对于有引用成员的类，合成拷贝赋值运算符被定义为删除的。

我们将在 13.6.2 节（第 476 页）、15.7.2 节（第 553 页）及 19.6 节（第 751 页）中介绍导致类的拷贝控制成员被定义为删除函数的其他原因。

本质上，当不可能拷贝、赋值或销毁类的成员时，类的合成拷贝控制成员就被定义为删除的。

private 拷贝控制

在新标准发布之前，类是通过将其拷贝构造函数和拷贝赋值运算符声明为 private 的来阻止拷贝：

```
class PrivateCopy {
    // 无访问说明符；接下来的成员默认为 private 的；参见 7.2 节（第 240 页）
    // 拷贝控制成员是 private 的，因此普通用户代码无法访问
    PrivateCopy(const PrivateCopy&);
    PrivateCopy &operator=(const PrivateCopy&);
    // 其他成员
public:
    PrivateCopy() = default; // 使用合成的默认构造函数
    ~PrivateCopy(); // 用户可以定义此类型的对象，但无法拷贝它们
};
```

由于析构函数是 public 的，用户可以定义 PrivateCopy 类型的对象。但是，由于拷贝构造函数和拷贝赋值运算符是 private 的，用户代码将不能拷贝这个类型的对象。但是，友元和成员函数仍旧可以拷贝对象。为了阻止友元和成员函数进行拷贝，我们将这些拷贝控制成员声明为 private 的，但并不定义它们。

声明但不定义一个成员函数是合法的（参见 6.1.2 节，第 186 页），对此只有一个例外，我们将在 15.2.1 节（第 528 页）中介绍。试图访问一个未定义的成员将导致一个链接时错误。通过声明（但不定义）private 的拷贝构造函数，我们可以预先阻止任何拷贝该类型对象的企图：试图拷贝对象的用户代码将在编译阶段被标记为错误；成员函数或友元函数中的拷贝操作将会导致链接时错误。 510

希望阻止拷贝的类应该使用=delete 来定义它们自己的拷贝构造函数和拷贝赋值运算符，而不应该将它们声明为 private 的。

13.1.6 节练习

练习 13.18：定义一个 Employee 类，它包含雇员的姓名和唯一的雇员证号。为这个类定义默认构造函数，以及接受一个表示雇员姓名的 string 的构造函数。每个构造函数应该通过递增一个 static 数据成员来生成一个唯一的证号。

练习 13.19：你的 Employee 类需要定义它自己的拷贝控制成员吗？如果需要，为什么？如果不需要，为什么？实现你认为 Employee 需要的拷贝控制成员。

练习 13.20：解释当我们拷贝、赋值或销毁 TextQuery 和 QueryResult 类（参见 12.3 节，第 430 页）对象时会发生什么。

练习 13.21：你认为 TextQuery 和 QueryResult 类需要定义它们自己版本的拷贝控制成员吗？如果需要，为什么？如果不需要，为什么？实现你认为这两个类需要的拷贝控制操作。

13.2 拷贝控制和资源管理

通常，管理类外资源的类必须定义拷贝控制成员。如我们在 13.1.4 节（第 447 页）中所见，这种类需要通过析构函数来释放对象所分配的资源。一旦一个类需要析构函数，那么它几乎肯定也需要一个拷贝构造函数和一个拷贝赋值运算符。

为了定义这些成员，我们首先必须确定此类型对象的拷贝语义。一般来说，有两种选择：可以定义拷贝操作，使类的行为看起来像一个值或者像一个指针。

类的行为像一个值，意味着它应该也有自己的状态。当我们拷贝一个像值的对象时，副本和原对象是完全独立的。改变副本不会对原对象有任何影响，反之亦然。

行为像指针的类则共享状态。当我们拷贝一个这种类的对象时，副本和原对象使用相同的底层数据。改变副本也会改变原对象，反之亦然。

在我们使用过的标准库类中，标准库容器和 string 类的行为像一个值。而不出意外的，shared_ptr 类提供类似指针的行为，就像我们的 StrBlob 类（参见 12.1.1 节，第 405 页）一样，IO 类型和 unique_ptr 不允许拷贝或赋值，因此它们的行为既不像值也不像指针。

511

为了说明这两种方式，我们会为练习中的 HasPtr 类定义拷贝控制成员。首先，我们将令类的行为像一个值；然后重新实现类，使它的行为像一个指针。

我们的 HasPtr 类有两个成员，一个 int 和一个 string 指针。通常，类直接拷贝内置类型（不包括指针）成员；这些成员本身就是值，因此通常应该让它们的行为像值一样。我们如何拷贝指针成员决定了像 HasPtr 这样的类是具有类值行为还是类指针行为。

13.2 节练习

练习 13.22：假定我们希望 HasPtr 的行为像一个值。即，对于对象所指向的 string

成员，每个对象都有一份自己的拷贝。我们将在下一节介绍拷贝控制成员的定义。但是，你已经学习了定义这些成员所需的所有知识。在继续学习下一节之前，为 HasPtr 编写拷贝构造函数和拷贝赋值运算符。

13.2.1 行为像值的类

为了提供类值的行为，对于类管理的资源，每个对象都应该拥有一份自己的拷贝。这意味着对于 ps 指向的 string，每个 HasPtr 对象都必须有自己的拷贝。为了实现类值行为，HasPtr 需要

- 定义一个拷贝构造函数，完成 string 的拷贝，而不是拷贝指针
- 定义一个析构函数来释放 string
- 定义一个拷贝赋值运算符来释放对象当前的 string，并从右侧运算对象拷贝 string

类值版本的 HasPtr 如下所示

```
class HasPtr {
public:
    HasPtr(const std::string &s = std::string()):
        ps(new std::string(s)), i(0) { }
    // 对 ps 指向的 string，每个 HasPtr 对象都有自己的拷贝
    HasPtr(const HasPtr &p):
        ps(new std::string(*p.ps)), i(p.i) { }
    HasPtr& operator=(const HasPtr &);
    ~HasPtr() { delete ps; }
private:
    std::string *ps;
    int     i;
};
```

我们的类足够简单，在类内就已定义了除赋值运算符之外的所有成员函数。第一个构造函数接受一个（可选的）string 参数。这个构造函数动态分配它自己的 string 副本，并将指向 string 的指针保存在 ps 中。拷贝构造函数也分配它自己的 string 副本。析构函数对指针成员 ps 执行 delete，释放构造函数中分配的内存。 512

类值拷贝赋值运算符

赋值运算符通常组合了析构函数和构造函数的操作。类似析构函数，赋值操作会销毁左侧运算对象的资源。类似拷贝构造函数，赋值操作会从右侧运算对象拷贝数据。但是，非常重要的一点是，这些操作是以正确的顺序执行的，即使将一个对象赋予它自身，也保证正确。而且，如果可能，我们编写的赋值运算符还应该是异常安全的——当异常发生时能将左侧运算对象置于一个有意义的状态（参见 5.6.2 节，第 175 页）。

在本例中，通过先拷贝右侧运算对象，我们可以处理自赋值情况，并能保证在异常发生时代码也是安全的。在完成拷贝后，我们释放左侧运算对象的资源，并更新指针指向新分配的 string：

```
HasPtr& HasPtr::operator=(const HasPtr &rhs)
{
    auto newp = new string(*rhs.ps); // 拷贝底层 string
    delete ps;        // 释放旧内存
```

```
    ps = newp;          // 从右侧运算对象拷贝数据到本对象
    i = rhs.i;
    return *this;       // 返回本对象
}
```

在这个赋值运算符中，非常清楚，我们首先进行了构造函数的工作：newp 的初始化器等价于 HasPtr 的拷贝构造函数中 ps 的初始化器。接下来与析构函数一样，我们 delete 当前 ps 指向的 string。然后就只剩下拷贝指向新分配的 string 的指针，以及从 rhs 拷贝 int 值到本对象了。

关键概念：赋值运算符

当你编写赋值运算符时，有两点需要记住：

- 如果将一个对象赋予它自身，赋值运算符必须能正确工作。
- 大多数赋值运算符组合了析构函数和拷贝构造函数的工作。

当你编写一个赋值运算符时，一个好的模式是先将右侧运算对象拷贝到一个局部临时对象中。当拷贝完成后，销毁左侧运算对象的现有成员就是安全的了。一旦左侧运算对象的资源被销毁，就只剩下将数据从临时对象拷贝到左侧运算对象的成员中了。

513 为了说明防范自赋值操作的重要性，考虑如果赋值运算符如下编写将会发生什么

```
// 这样编写赋值运算符是错误的！
HasPtr&
HasPtr::operator=(const HasPtr &rhs)
{
    delete ps; // 释放对象指向的 string
    // 如果 rhs 和*this 是同一个对象，我们就将从已释放的内存中拷贝数据！
    ps = new string(*(rhs.ps));
    i = rhs.i;
    return *this;
}
```

如果 rhs 和本对象是同一个对象，delete ps 会释放*this 和 rhs 指向的 string。接下来，当我们在 new 表达式中试图拷贝*(rhs.ps)时，就会访问一个指向无效内存的指针，其行为和结果是未定义的。

WARNING

对于一个赋值运算符来说，正确工作是非常重要的，即使是将一个对象赋予它自身，也要能正确工作。一个好的方法是在销毁左侧运算对象资源之前拷贝右侧运算对象。

13.2.1 节练习

练习 13.23：比较上一节练习中你编写的拷贝控制成员和这一节中的代码。确定你理解了你的代码和我们的代码之间的差异（如果有的话）。

练习 13.24：如果本节中的 HasPtr 版本未定义析构函数，将会发生什么？如果未定义拷贝构造函数，将会发生什么？

练习 13.25：假定希望定义 StrBlob 的类值版本，而且希望继续使用 shared_ptr，

这样我们的 StrBlobPtr 类就仍能使用指向 vector 的 weak_ptr 了。你修改后的类将需要一个拷贝构造函数和一个拷贝赋值运算符，但不需要析构函数。解释拷贝构造函数和拷贝赋值运算符必须要做什么。解释为什么不需要析构函数。

练习 13.26：对上一题中描述的 StrBlob 类，编写你自己的版本。

13.2.2 定义行为像指针的类

对于行为类似指针的类，我们需要为其定义拷贝构造函数和拷贝赋值运算符，来拷贝指针成员本身而不是它指向的 string。我们的类仍然需要自己的析构函数来释放接受 string 参数的构造函数分配的内存（参见 13.1.4 节，第 447 页）。但是，在本例中，析构函数不能单方面地释放关联的 string。只有当最后一个指向 string 的 HasPtr 销毁时，它才可以释放 string。

令一个类展现类似指针的行为的最好方法是使用 shared_ptr 来管理类中的资源。拷贝（或赋值）一个 shared_ptr 会拷贝（赋值）shared_ptr 所指向的指针。shared_ptr 类自己记录有多少用户共享它所指向的对象。当没有用户使用对象时，shared_ptr 类负责释放资源。 514

但是，有时我们希望直接管理资源。在这种情况下，使用**引用计数**（reference count）（参见 12.1.1 节，第 402 页）就很有用了。为了说明引用计数如何工作，我们将重新定义 HasPtr，令其行为像指针一样，但我们不使用 shared_ptr，而是设计自己的引用计数。

引用计数

引用计数的工作方式如下：

- 除了初始化对象外，每个构造函数（拷贝构造函数除外）还要创建一个引用计数，用来记录有多少对象与正在创建的对象共享状态。当我们创建一个对象时，只有一个对象共享状态，因此将计数器初始化为 1。
- 拷贝构造函数不分配新的计数器，而是拷贝给定对象的数据成员，包括计数器。拷贝构造函数递增共享的计数器，指出给定对象的状态又被一个新用户所共享。
- 析构函数递减计数器，指出共享状态的用户少了一个。如果计数器变为 0，则析构函数释放状态。
- 拷贝赋值运算符递增右侧运算对象的计数器，递减左侧运算对象的计数器。如果左侧运算对象的计数器变为 0，意味着它的共享状态没有用户了，拷贝赋值运算符就必须销毁状态。

唯一的难题是确定在哪里存放引用计数。计数器不能直接作为 HasPtr 对象的成员。下面的例子说明了原因：

```
HasPtr p1("Hiya!");
HasPtr p2(p1);    // p1 和 p2 指向相同的 string
HasPtr p3(p1);    // p1、p2 和 p3 都指向相同的 string
```

如果引用计数保存在每个对象中，当创建 p3 时我们应该如何正确更新它呢？可以递增 p1 中的计数器并将其拷贝到 p3 中，但如何更新 p2 中的计数器呢？

解决此问题的一种方法是将计数器保存在动态内存中。当创建一个对象时，我们也分配一个新的计数器。当拷贝或赋值对象时，我们拷贝指向计数器的指针。使用这种方法，副本和原对象都会指向相同的计数器。

定义一个使用引用计数的类

通过使用引用计数，我们就可以编写类指针的 HasPtr 版本了：

515

```
class HasPtr {
public:
    // 构造函数分配新的 string 和新的计数器，将计数器置为 1
    HasPtr(const std::string &s = std::string()):
      ps(new std::string(s)), i(0), use(new std::size_t(1)) {}
    // 拷贝构造函数拷贝所有三个数据成员，并递增计数器
    HasPtr(const HasPtr &p):
        ps(p.ps), i(p.i), use(p.use) { ++*use; }
    HasPtr& operator=(const HasPtr&);
    ~HasPtr();
private:
    std::string *ps;
    int i;
    std::size_t *use; // 用来记录有多少个对象共享*ps 的成员
};
```

在此，我们添加了一个名为 use 的数据成员，它记录有多少对象共享相同的 string。接受 string 参数的构造函数分配新的计数器，并将其初始化为 1，指出当前有一个用户使用本对象的 string 成员。

类指针的拷贝成员“篡改”引用计数

当拷贝或赋值一个 HasPtr 对象时，我们希望副本和原对象都指向相同的 string。即，当拷贝一个 HasPtr 时，我们将拷贝 ps 本身，而不是 ps 指向的 string。当我们进行拷贝时，还会递增该 string 关联的计数器。

（我们在类内定义的）拷贝构造函数拷贝给定 HasPtr 的所有三个数据成员。这个构造函数还递增 use 成员，指出 ps 和 p.ps 指向的 string 又有了一个新的用户。

析构函数不能无条件地 delete ps——可能还有其他对象指向这块内存。析构函数应该递减引用计数，指出共享 string 的对象少了一个。如果计数器变为 0，则析构函数释放 ps 和 use 指向的内存：

```
HasPtr::~HasPtr()
{
    if (--*use == 0) {    // 如果引用计数变为 0
        delete ps;        // 释放 string 内存
        delete use;       // 释放计数器内存
    }
}
```

拷贝赋值运算符与往常一样执行类似拷贝构造函数和析构函数的工作。即，它必须递增右侧运算对象的引用计数（即，拷贝构造函数的工作），并递减左侧运算对象的引用计数，在必要时释放使用的内存（即，析构函数的工作）。

516

而且与往常一样，赋值运算符必须处理自赋值。我们通过先递增 rhs 中的计数然后再递减左侧运算对象中的计数来实现这一点。通过这种方法，当两个对象相同时，在我们检查 ps（及 use）是否应该释放之前，计数器就已经被递增过了：

```
HasPtr& HasPtr::operator=(const HasPtr &rhs)
{
```

```
    ++*rhs.use; // 递增右侧运算对象的引用计数
    if (--*use == 0) {    // 然后递减本对象的引用计数
        delete ps;        // 如果没有其他用户
        delete use;       // 释放本对象分配的成员
    }
    ps = rhs.ps;          // 将数据从 rhs 拷贝到本对象
    i = rhs.i;
    use = rhs.use;
    return *this;         // 返回本对象
}
```

13.2.2 节练习

练习 13.27：定义你自己的使用引用计数版本的 HasPtr。

练习 13.28：给定下面的类，为其实现一个默认构造函数和必要的拷贝控制成员。

```
(a) class TreeNode {
    private:
        std::string value;
        int         count;
        TreeNode    *left;
        TreeNode    *right;
    };
```

```
(b) class BinStrTree {
    private:
        TreeNode *root;
    };
```

13.3 交换操作

除了定义拷贝控制成员，管理资源的类通常还定义一个名为 swap 的函数（参见 9.2.5 节，第 303 页）。对于那些与重排元素顺序的算法（参见 10.2.3 节，第 342 页）一起使用的类，定义 swap 是非常重要的。这类算法在需要交换两个元素时会调用 swap。

如果一个类定义了自己的 swap，那么算法将使用类自定义版本。否则，算法将使用标准库定义的 swap。虽然与往常一样我们不知道 swap 是如何实现的，但理论上很容易理解，为了交换两个对象我们需要进行一次拷贝和两次赋值。例如，交换两个类值 HasPtr 对象（参见 13.2.1 节，第 453 页）的代码可能像下面这样：

```
HasPtr temp = v1;     // 创建 v1 的值的一个临时副本
v1 = v2;              // 将 v2 的值赋予 v1
v2 = temp;            // 将保存的 v1 的值赋予 v2
```

这段代码将原来 v1 中的 string 拷贝了两次——第一次是 HasPtr 的拷贝构造函数将 v1 拷贝给 temp，第二次是赋值运算符将 temp 赋予 v2。将 v2 赋予 v1 的语句还拷贝了原来 v2 中的 string。如我们所见，拷贝一个类值的 HasPtr 会分配一个新 string 并将其拷贝到 HasPtr 指向的位置。 517

理论上，这些内存分配都是不必要的。我们更希望 swap 交换指针，而不是分配 string 的新副本。即，我们希望这样交换两个 HasPtr：

```
string *temp = v1.ps;     // 为 v1.ps 中的指针创建一个副本
v1.ps = v2.ps;            // 将 v2.ps 中的指针赋予 v1.ps
v2.ps = temp;             // 将保存的 v1.ps 中原来的指针赋予 v2.ps
```

编写我们自己的 swap 函数

可以在我们的类上定义一个自己版本的 swap 来重载 swap 的默认行为。swap 的典型实现如下：

```
class HasPtr {
    friend void swap(HasPtr&, HasPtr&);
    // 其他成员定义，与 13.2.1 节（第 453 页）中一样
};
inline
void swap(HasPtr &lhs, HasPtr &rhs)
{
    using std::swap;
    swap(lhs.ps, rhs.ps);      // 交换指针，而不是 string 数据
    swap(lhs.i, rhs.i);        // 交换 int 成员
}
```

我们首先将 swap 定义为 friend，以便能访问 HasPtr 的（private 的）数据成员。由于 swap 的存在就是为了优化代码，我们将其声明为 inline 函数（参见 6.5.2 节，第 213 页）。swap 的函数体对给定对象的每个数据成员调用 swap。我们首先 swap 绑定到 rhs 和 lhs 的对象的指针成员，然后是 int 成员。

与拷贝控制成员不同，swap 并不是必要的。但是，对于分配了资源的类，定义 swap 可能是一种很重要的优化手段。

swap 函数应该调用 swap，而不是 std::swap

此代码中有一个很重要的微妙之处：虽然这一点在这个特殊的例子中并不重要，但在一般情况下它非常重要——swap 函数中调用的 swap 不是 std::swap。在本例中，数据成员是内置类型的，而内置类型是没有特定版本的 swap 的，所以在本例中，对 swap 的调用会调用标准库 std::swap。

但是，如果一个类的成员有自己类型特定的 swap 函数，调用 std::swap 就是错误的了。例如，假定我们有另一个命名为 Foo 的类，它有一个类型为 HasPtr 的成员 h。如果我们未定义 Foo 版本的 swap，那么就会使用标准库版本的 swap。如我们所见，标准库 swap 对 HasPtr 管理的 string 进行了不必要的拷贝。

518

我们可以为 Foo 编写一个 swap 函数，来避免这些拷贝。但是，如果这样编写 Foo 版本的 swap：

```
void swap(Foo &lhs, Foo &rhs)
{
    // 错误：这个函数使用了标准库版本的 swap，而不是 HasPtr 版本
    std::swap(lhs.h, rhs.h);
    // 交换类型 Foo 的其他成员
}
```

此编码会编译通过，且正常运行。但是，使用此版本与简单使用默认版本的 swap 并没有任何性能差异。问题在于我们显式地调用了标准库版本的 swap。但是，我们不希望使用 std 中的版本，我们希望调用为 HasPtr 对象定义的版本。

正确的 swap 函数如下所示：

```
void swap(Foo &lhs, Foo &rhs)
```

```
{
    using std::swap;
    swap(lhs.h, rhs.h); // 使用 HasPtr 版本的 swap
    // 交换类型 Foo 的其他成员
}
```

每个 swap 调用应该都是未加限定的。即，每个调用都应该是 swap，而不是 std::swap。如果存在类型特定的 swap 版本，其匹配程度会优于 std 中定义的版本，原因我们将在 16.3 节（第 616 页）中进行解释。因此，如果存在类型特定的 swap 版本，swap 调用会与之匹配。如果不存在类型特定的版本，则会使用 std 中的版本（假定作用域中有 using 声明）。

非常仔细的读者可能会奇怪为什么 swap 函数中的 using 声明没有隐藏 HasPtr 版本 swap 的声明（参见 6.4.1 节，第 210 页）。我们将在 18.2.3 节（第 706 页）中解释为什么这段代码能正常工作。

在赋值运算符中使用 swap

定义 swap 的类通常用 swap 来定义它们的赋值运算符。这些运算符使用了一种名为**拷贝并交换**（copy and swap）的技术。这种技术将左侧运算对象与右侧运算对象的一个副本进行交换：

```
// 注意 rhs 是按值传递的，意味着 HasPtr 的拷贝构造函数
// 将右侧运算对象中的 string 拷贝到 rhs
HasPtr& HasPtr::operator=(HasPtr rhs)
{
    // 交换左侧运算对象和局部变量 rhs 的内容
    swap(*this, rhs);     // rhs 现在指向本对象曾经使用的内存
    return *this;         // rhs 被销毁，从而 delete 了 rhs 中的指针
}
```

在这个版本的赋值运算符中，参数并不是一个引用，我们将右侧运算对象以传值方式传递给了赋值运算符。因此，rhs 是右侧运算对象的一个副本。参数传递时拷贝 HasPtr 的操作会分配该对象的 string 的一个新副本。 519

在赋值运算符的函数体中，我们调用 swap 来交换 rhs 和*this 中的数据成员。这个调用将左侧运算对象中原来保存的指针存入 rhs 中，并将 rhs 中原来的指针存入*this 中。因此，在 swap 调用之后，*this 中的指针成员将指向新分配的 string——右侧运算对象中 string 的一个副本。

当赋值运算符结束时，rhs 被销毁，HasPtr 的析构函数将执行。此析构函数 delete rhs 现在指向的内存，即，释放掉左侧运算对象中原来的内存。

这个技术的有趣之处是它自动处理了自赋值情况且天然就是异常安全的。它通过在改变左侧运算对象之前拷贝右侧运算对象保证了自赋值的正确，这与我们在原来的赋值运算符中使用的方法是一致的（参见 13.2.1 节，第 453 页）。它保证异常安全的方法也与原来的赋值运算符实现一样。代码中唯一可能抛出异常的是拷贝构造函数中的 new 表达式。如果真发生了异常，它也会在我们改变左侧运算对象之前发生。

使用拷贝和交换的赋值运算符自动就是异常安全的，且能正确处理自赋值。

13.3 节练习

练习 13.29：解释 swap(HasPtr&, HasPtr&)中对 swap 的调用不会导致递归循环。

练习 13.30：为你的类值版本的 HasPtr 编写 swap 函数，并测试它。为你的 swap 函数添加一个打印语句，指出函数什么时候执行。

练习 13.31：为你的 HasPtr 类定义一个<运算符，并定义一个 HasPtr 的 vector。为这个 vector 添加一些元素，并对它执行 sort。注意何时会调用 swap。

练习 13.32：类指针的 HasPtr 版本会从 swap 函数受益吗？如果会，得到了什么益处？如果不是，为什么？

13.4 拷贝控制示例

虽然通常来说分配资源的类更需要拷贝控制，但资源管理并不是一个类需要定义自己的拷贝控制成员的唯一原因。一些类也需要拷贝控制成员的帮助来进行簿记工作或其他操作。

作为类需要拷贝控制来进行簿记操作的例子，我们将概述两个类的设计，这两个类可能用于邮件处理应用中。两个类命名为 Message 和 Folder，分别表示电子邮件（或者其他类型的）消息和消息目录。每个 Message 对象可以出现在多个 Folder 中。但是，任意给定的 Message 的内容只有一个副本。这样，如果一条 Message 的内容被改变，则我们从它所在的任何 Folder 来浏览此 Message 时，都会看到改变后的内容。

520

为了记录 Message 位于哪些 Folder 中，每个 Message 都会保存一个它所在 Folder 的指针的 set，同样的，每个 Folder 都保存一个它包含的 Message 的指针的 set。图 13.1 说明了这种设计思路。

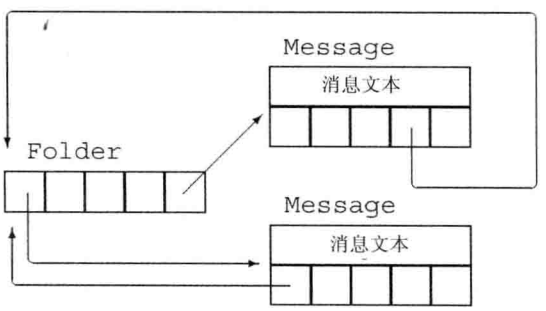


图 13.1：Message 和 Folder 类设计

我们的 Message 类会提供 save 和 remove 操作，来向一个给定 Folder 添加一条 Message 或是从中删除一条 Message。为了创建一个新的 Message，我们会指明消息内容，但不会指出 Folder。为了将一条 Message 放到一个特定 Folder 中，我们必须调用 save。

当我们拷贝一个 Message 时，副本和原对象将是不同的 Message 对象，但两个 Message 都出现在相同的 Folder 中。因此，拷贝 Message 的操作包括消息内容和 Folder 指针 set 的拷贝。而且，我们必须在每个包含此消息的 Folder 中都添加一个指向新创建的 Message 的指针。

当我们销毁一个 Message 时，它将不复存在。因此，我们必须从包含此消息的所有

Folder 中删除指向此 Message 的指针。

当我们将一个 Message 对象赋予另一个 Message 对象时，左侧 Message 的内容会被右侧 Message 的内容所替代。我们还必须更新 Folder 集合，从原来包含左侧 Message 的 Folder 中将它删除，并将它添加到包含右侧 Message 的 Folder 中。

观察这些操作，我们可以看到，析构函数和拷贝赋值运算符都必须从包含一条 Message 的所有 Folder 中删除它。类似的，拷贝构造函数和拷贝赋值运算符都要将一个 Message 添加到给定的一组 Folder 中。我们将定义两个 private 的工具函数来完成这些工作。

拷贝赋值运算符通常执行拷贝构造函数和析构函数中也要做的工作。这种情况下，公共的工作应该放在 private 的工具函数中完成。

Folder 类也需要类似的拷贝控制成员，来添加或删除它保存的 Message。 521

我们将 Folder 类的设计和实现留作练习。但是，我们将假定 Folder 类包含名为 addMsg 和 remMsg 的成员，分别完成在给定 Folder 对象的消息集合中添加和删除 Message 的工作。

Message 类

根据上述设计，我们可以编写 Message 类，如下所示：

```
class Message {
    friend class Folder;
public:
    // folders 被隐式初始化为空集合
    explicit Message(const std::string &str = ""):
        contents(str) { }
    // 拷贝控制成员，用来管理指向本 Message 的指针
    Message(const Message&);                // 拷贝构造函数
    Message& operator=(const Message&);     // 拷贝赋值运算符
    ~Message();                             // 析构函数
    // 从给定 Folder 集合中添加/删除本 Message
    void save(Folder&);
    void remove(Folder&);
private:
    std::string contents;          // 实际消息文本
    std::set<Folder*> folders;     // 包含本 Message 的 Folder
    // 拷贝构造函数、拷贝赋值运算符和析构函数所使用的工具函数
    // 将本 Message 添加到指向参数的 Folder 中
    void add_to_Folders(const Message&);
    // 从 folders 中的每个 Folder 中删除本 Message
    void remove_from_Folders();
};
```

这个类定义了两个数据成员：contents，保存消息文本；folders，保存指向本 Message 所在 Folder 的指针。接受一个 string 参数的构造函数将给定 string 拷贝给 contents，并将 folders（隐式）初始化为空集。由于此构造函数有一个默认参数，因此它也被当作 Message 的默认构造函数（参见 7.5.1 节，第 260 页）。

save 和 remove 成员

除拷贝控制成员外，Message 类只有两个公共成员：save，将本 Message 存放在给定 Folder 中；remove，删除本 Message：

```
void Message::save(Folder &f)
{
    folders.insert(&f);   // 将给定 Folder 添加到我们的 Folder 列表中
    f.addMsg(this);       // 将本 Message 添加到 f 的 Message 集合中
}
void Message::remove(Folder &f)
{
    folders.erase(&f);    // 将给定 Folder 从我们的 Folder 列表中删除
    f.remMsg(this);       // 将本 Message 从 f 的 Message 集合中删除
}
```

522

为了保存（或删除）一个 Message，需要更新本 Message 的 folders 成员。当 save 一个 Message 时，我们应保存一个指向给定 Folder 的指针；当 remove 一个 Message 时，我们要删除此指针。

这些操作还必须更新给定的 Folder。更新一个 Folder 的任务是由 Folder 类的 addMsg 和 remMsg 成员来完成的，分别添加和删除给定 Message 的指针。

Message 类的拷贝控制成员

当我们拷贝一个 Message 时，得到的副本应该与原 Message 出现在相同的 Folder 中。因此，我们必须遍历 Folder 指针的 set，对每个指向原 Message 的 Folder 添加一个指向新 Message 的指针。拷贝构造函数和拷贝赋值运算符都需要做这个工作，因此我们定义一个函数来完成这个公共操作：

```
// 将本 Message 添加到指向 m 的 Folder 中
void Message::add_to_Folders(const Message &m)
{
    for (auto f : m.folders) // 对每个包含 m 的 Folder
        f->addMsg(this);     // 向该 Folder 添加一个指向本 Message 的指针
}
```

此例中我们对 m.folders 中每个 Folder 调用 addMsg。函数 addMsg 会将本 Message 的指针添加到每个 Folder 中。

Message 的拷贝构造函数拷贝给定对象的数据成员：

```
Message::Message(const Message &m):
    contents(m.contents), folders(m.folders)
{
    add_to_Folders(m); // 将本消息添加到指向 m 的 Folder 中
}
```

并调用 add_to_Folders 将新创建的 Message 的指针添加到每个包含原 Message 的 Folder 中。

Message 的析构函数

当一个 Message 被销毁时，我们必须从指向此 Message 的 Folder 中删除它。拷贝赋值运算符也要执行此操作，因此我们会定义一个公共函数来完成此工作：

```
// 从对应的 Folder 中删除本 Message
void Message::remove_from_Folders()
{
    for (auto f : folders)    // 对 folders 中每个指针
        f->remMsg(this);      // 从该 Folder 中删除本 Message
}
```

函数 remove_from_Folders 的实现类似 add_to_Folders，不同之处是它调用 remMsg 来删除当前 Message 而不是调用 addMsg 来添加 Message。 <523>

有了 remove_from_Folders 函数，编写析构函数就很简单了：

```
Message::~Message()
{
    remove_from_Folders();
}
```

调用 remove_from_Folders 确保没有任何 Folder 保存正在销毁的 Message 的指针。编译器自动调用 string 的析构函数来释放 contents，并自动调用 set 的析构函数来清理集合成员使用的内存。

Message 的拷贝赋值运算符

与大多数赋值运算符相同，我们的 Message 类的拷贝赋值运算符必须执行拷贝构造函数和析构函数的工作。与往常一样，最重要的是我们要组织好代码结构，使得即使左侧和右侧运算对象是同一个 Message，拷贝赋值运算符也能正确执行。

在本例中，我们先从左侧运算对象的 folders 中删除此 Message 的指针，然后再将指针添加到右侧运算对象的 folders 中，从而实现了自赋值的正确处理：

```
Message& Message::operator=(const Message &rhs)
{
    // 通过先删除指针再插入它们来处理自赋值情况
    remove_from_Folders();    // 更新已有 Folder
    contents = rhs.contents;  // 从 rhs 拷贝消息内容
    folders = rhs.folders;    // 从 rhs 拷贝 Folder 指针
    add_to_Folders(rhs);      // 将本 Message 添加到那些 Folder 中
    return *this;
}
```

如果左侧和右侧运算对象是相同的 Message，则它们具有相同的地址。如果我们在 add_to_Folders 之后调用 remove_from_Folders，就会将此 Message 从它所在的所有 Folder 中删除。

Message 的 swap 函数

标准库中定义了 string 和 set 的 swap 版本（参见 9.2.5 节，第 303 页）。因此，如果为我们的 Message 类定义它自己的 swap 版本，它将从中受益。通过定义一个 Message 特定版本的 swap，我们可以避免对 contents 和 folders 成员进行不必要的拷贝。

但是，我们的 swap 函数必须管理指向被交换 Message 的 Folder 指针。在调用 swap(m1,m2) 之后，原来指向 m1 的 Folder 现在必须指向 m2，反之亦然。

我们通过两遍扫描 folders 中每个成员来正确处理 Folder 指针。第一遍扫描将 Message 从它们所在的 Folder 中删除。接下来我们调用 swap 来交换数据成员。最后

对 folders 进行第二遍扫描来添加交换过的 Message：

524

```
void swap(Message &lhs, Message &rhs)
{
    using std::swap; // 在本例中严格来说并不需要，但这是一个好习惯
    // 将每个消息的指针从它（原来）所在 Folder 中删除
    for (auto f: lhs.folders)
        f->remMsg(&lhs);
    for (auto f: rhs.folders)
        f->remMsg(&rhs);
    // 交换 contents 和 Folder 指针 set
    swap(lhs.folders, rhs.folders);        // 使用 swap(set&, set&)
    swap(lhs.contents, rhs.contents);      // swap(string&, string&)
    // 将每个 Message 的指针添加到它的（新）Folder 中
    for (auto f: lhs.folders)
        f->addMsg(&lhs);
    for (auto f: rhs.folders)
        f->addMsg(&rhs);
}
```

13.4 节练习

练习 13.33：为什么 Message 的成员 save 和 remove 的参数是一个 Folder&？为什么我们不将参数定义为 Folder 或是 const Folder&？

练习 13.34：编写本节所描述的 Message。

练习 13.35：如果 Message 使用合成的拷贝控制成员，将会发生什么？

练习 13.36：设计并实现对应的 Folder 类。此类应该保存一个指向 Folder 中包含的 Message 的 set。

练习 13.37：为 Message 类添加成员，实现向 folders 添加或删除一个给定的 Folder*。这两个成员类似 Folder 类的 addMsg 和 remMsg 操作。

练习 13.38：我们并未使用拷贝和交换方式来设计 Message 的赋值运算符。你认为其原因是什么？

13.5 动态内存管理类

某些类需要在运行时分配可变大小的内存空间。这种类通常可以（并且如果它们确实可以的话，一般应该）使用标准库容器来保存它们的数据。例如，我们的 StrBlob 类使用一个 vector 来管理其元素的底层内存。

但是，这一策略并不是对每个类都适用；某些类需要自己进行内存分配。这些类一般来说必须定义自己的拷贝控制成员来管理所分配的内存。

525

例如，我们将实现标准库 vector 类的一个简化版本。我们所做的一个简化是不使用模板，我们的类只用于 string。因此，它被命名为 StrVec。

StrVec 类的设计

回忆一下，vector 类将其元素保存在连续内存中。为了获得可接受的性能，vector

预先分配足够的内存来保存可能需要的更多元素（参见 9.4 节，第 317 页）。vector 的每个添加元素的成员函数会检查是否有空间容纳更多的元素。如果有，成员函数会在下一个可用位置构造一个对象。如果没有可用空间，vector 就会重新分配空间：它获得新的空间，将已有元素移动到新空间中，释放旧空间，并添加新元素。

我们在 StrVec 类中使用类似的策略。我们将使用一个 allocator 来获得原始内存（参见 12.2.2 节，第 427 页）。由于 allocator 分配的内存是未构造的，我们将在需要添加新元素时用 allocator 的 construct 成员在原始内存中创建对象。类似的，当我们需要删除一个元素时，我们将使用 destroy 成员来销毁元素。

每个 StrVec 有三个指针成员指向其元素所使用的内存：

- elements，指向分配的内存中的首元素
- first_free，指向最后一个实际元素之后的位置
- cap，指向分配的内存末尾之后的位置

图 13.2 说明了这些指针的含义。

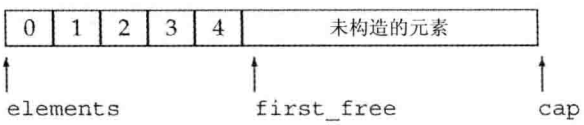


图 13.2：StrVec 内存分配策略

除了这些指针之外，StrVec 还有一个名为 alloc 的静态成员，其类型为 allocator<string>。alloc 成员会分配 StrVec 使用的内存。我们的类还有 4 个工具函数：

- alloc_n_copy 会分配内存，并拷贝一个给定范围中的元素。
- free 会销毁构造的元素并释放内存。
- chk_n_alloc 保证 StrVec 至少有容纳一个新元素的空间。如果没有空间添加新元素，chk_n_alloc 会调用 reallocate 来分配更多内存。
- reallocate 在内存用完时为 StrVec 分配新内存。

虽然我们关注的是类的实现，但我们也将定义 vector 接口中的一些成员。

StrVec 类定义

526

有了上述实现概要，我们现在可以定义 StrVec 类，如下所示：

```
// 类 vector 类内存分配策略的简化实现
class StrVec {
public:
    StrVec(): // allocator 成员进行默认初始化
      elements(nullptr), first_free(nullptr), cap(nullptr) { }
    StrVec(const StrVec&);                  // 拷贝构造函数
    StrVec &operator=(const StrVec&);       // 拷贝赋值运算符
    ~StrVec();                              // 析构函数
    void push_back(const std::string&);     // 拷贝元素
    size_t size() const { return first_free - elements; }
    size_t capacity() const { return cap - elements; }
    std::string *begin() const { return elements; }
    std::string *end() const { return first_free; }
```

```
    // ...
private:
    Static std::allocator<std::string> alloc; // 分配元素
    // 被添加元素的函数所使用
    void chk_n_alloc()
        { if (size() == capacity()) reallocate(); }
    // 工具函数，被拷贝构造函数、赋值运算符和析构函数所使用
    std::pair<std::string*, std::string*> alloc_n_copy
        (const std::string*, const std::string*);
    void free();                // 销毁元素并释放内存
    void reallocate();          // 获得更多内存并拷贝已有元素
    std::string *elements;      // 指向数组首元素的指针
    std::string *first_free;    // 指向数组第一个空闲元素的指针
    std::string *cap;           // 指向数组尾后位置的指针
};
```

类体定义了多个成员：

- 默认构造函数(隐式地)默认初始化 alloc 并(显式地)将指针初始化为 nullptr，表明没有元素。
- size 成员返回当前真正在使用的元素的数目，等于 first_free-elements。
- capacity 成员返回 StrVec 可以保存的元素的数量，等价于 cap-elements。
- 当没有空间容纳新元素，即 cap==first_free 时，chk_n_alloc 会为 StrVec 重新分配内存。
- begin 和 end 成员分别返回指向首元素（即 elements）和最后一个构造的元素之后位置（即 first_free）的指针。

使用 construct

函数 push_back 调用 chk_n_alloc 确保有空间容纳新元素。如果需要，chk_n_alloc 会调用 reallocate。当 chk_n_alloc 返回时，push_back 知道必有空间容纳新元素。它要求其 allocator 成员来 construct 新的尾元素：

527

```
void StrVec::push_back(const string& s)
{
    chk_n_alloc(); // 确保有空间容纳新元素
    // 在 first_free 指向的元素中构造 s 的副本
    alloc.construct(first_free++, s);
}
```

当我们用 allocator 分配内存时，必须记住内存是未构造的(参见 12.2.2 节，第 428 页)。为了使用此原始内存，我们必须调用 construct，在此内存中构造一个对象。传递给 construct 的第一个参数必须是一个指针，指向调用 allocate 所分配的未构造的内存空间。剩余参数确定用哪个构造函数来构造对象。在本例中，只有一个额外参数，类型为 string，因此会使用 string 的拷贝构造函数。

值得注意的是，对 construct 的调用也会递增 first_free，表示已经构造了一个新元素。它使用前置递增（参见 4.5 节，第 131 页），因此这个调用会在 first_free 当前值指定的地址构造一个对象，并递增 first_free 指向下一个未构造的元素。

alloc_n_copy 成员

我们在拷贝或赋值 StrVec 时，可能会调用 alloc_n_copy 成员。类似 vector，我们的 StrVec 类有类值的行为（参见 13.2.1 节，第 453 页）。当我们拷贝或赋值 StrVec 时，必须分配独立的内存，并从原 StrVec 对象拷贝元素至新对象。

alloc_n_copy 成员会分配足够的内存来保存给定范围的元素，并将这些元素拷贝到新分配的内存中。此函数返回一个指针的 pair（参见 11.2.3 节，第 379 页），两个指针分别指向新空间的开始位置和拷贝的尾后的位置：

```
pair<string*, string*>
StrVec::alloc_n_copy(const string *b, const string *e)
{
    // 分配空间保存给定范围中的元素
    auto data = alloc.allocate(e - b);
    // 初始化并返回一个 pair，该 pair 由 data 和 uninitialized_copy 的返回值构成
    return {data, uninitialized_copy(b, e, data)};
}
```

alloc_n_copy 用尾后指针减去首元素指针，来计算需要多少空间。在分配内存之后，它必须在此空间中构造给定元素的副本。

它是在返回语句中完成拷贝工作的，返回语句中对返回值进行了列表初始化（参见 6.3.2 节，第 203 页）。返回的 pair 的 first 成员指向分配的内存的开始位置；second 成员则是 uninitialized_copy（参见 12.2.2 节，第 429 页）的返回值，此值是一个指针，指向最后一个构造元素之后的位置。 528

free 成员

free 成员有两个责任：首先 destroy 元素，然后释放 StrVec 自己分配的内存空间。for 循环调用 allocator 的 destroy 成员，从构造的尾元素开始，到首元素为止，逆序销毁所有元素：

```
void StrVec::free()
{
    // 不能传递给 deallocate 一个空指针，如果 elements 为 0，函数什么也不做
    if (elements) {
        // 逆序销毁旧元素
        for (auto p = first_free; p != elements; /* 空 */)
            alloc.destroy(--p);
        alloc.deallocate(elements, cap - elements);
    }
}
```

destroy 函数会运行 string 的析构函数。string 的析构函数会释放 string 自己分配的内存空间。

一旦元素被销毁，我们就调用 deallocate 来释放本 StrVec 对象分配的内存空间。我们传递给 deallocate 的指针必须是之前某次 allocate 调用所返回的指针。因此，在调用 deallocate 之前我们首先检查 elements 是否为空。

拷贝控制成员

实现了 alloc_n_copy 和 free 成员后，为我们的类实现拷贝控制成员就很简单了。

拷贝构造函数调用 alloc_n_copy：

```
StrVec::StrVec(const StrVec &s)
{
    // 调用 alloc_n_copy 分配空间以容纳与 s 中一样多的元素
    auto newdata = alloc_n_copy(s.begin(), s.end());
    elements = newdata.first;
    first_free = cap = newdata.second;
}
```

并将返回结果赋予数据成员。alloc_n_copy 的返回值是一个指针的 pair。其 first 成员指向第一个构造的元素，second 成员指向最后一个构造的元素之后的位置。由于 alloc_n_copy 分配的空间恰好容纳给定的元素，cap 也指向最后一个构造的元素之后的位置。

析构函数调用 free：

```
StrVec::~StrVec() { free(); }
```

拷贝赋值运算符在释放已有元素之前调用 alloc_n_copy，这样就可以正确处理自赋值了：

529

```
StrVec &StrVec::operator=(const StrVec &rhs)
{
    // 调用 alloc_n_copy 分配内存，大小与 rhs 中元素占用空间一样多
    auto data = alloc_n_copy(rhs.begin(), rhs.end());
    free();
    elements = data.first;
    first_free = cap = data.second;
    return *this;
}
```

类似拷贝构造函数，拷贝赋值运算符使用 alloc_n_copy 的返回值来初始化它的指针。

在重新分配内存的过程中移动而不是拷贝元素

在编写 reallocate 成员函数之前，我们稍微思考一下此函数应该做什么。它应该

- 为一个新的、更大的 string 数组分配内存
- 在内存空间的前一部分构造对象，保存现有元素
- 销毁原内存空间中的元素，并释放这块内存

观察这个操作步骤，我们可以看出，为一个 StrVec 重新分配内存空间会引起从旧内存空间到新内存空间逐个拷贝 string。虽然我们不知道 string 的实现细节，但我们知道 string 具有类值行为。当拷贝一个 string 时，新 string 和原 string 是相互独立的。改变原 string 不会影响到副本，反之亦然。

由于 string 的行为类似值，我们可以得出结论，每个 string 对构成它的所有字符都会保存自己的一份副本。拷贝一个 string 必须为这些字符分配内存空间，而销毁一个 string 必须释放所占用的内存。

拷贝一个 string 就必须真的拷贝数据，因为通常情况下，在我们拷贝了一个 string 之后，它就会有两个用户。但是，如果是 reallocate 拷贝 StrVec 中的 string，则在拷贝之后，每个 string 只有唯一的用户。一旦将元素从旧空间拷贝到了新空间，我们就会立即销毁原 string。

因此，拷贝这些 string 中的数据是多余的。在重新分配内存空间时，如果我们能避免分配和释放 string 的额外开销，StrVec 的性能会好得多。

移动构造函数和 std::move

通过使用新标准库引入的两种机制，我们就可以避免 string 的拷贝。首先，有一些标准库类，包括 string，都定义了所谓的“移动构造函数”。关于 string 的移动构造函数如何工作的细节，以及有关实现的任何其他细节，目前都尚未公开。但是，我们知道，移动构造函数通常是将资源从给定对象“移动”而不是拷贝到正在创建的对象。而且我们知道标准库保证“移后源”（moved-from）string 仍然保持一个有效的、可析构的状态。对于 string，我们可以想象每个 string 都有一个指向 char 数组的指针。可以假定 string 的移动构造函数进行了指针的拷贝，而不是为字符分配内存空间然后拷贝字符。

C++ 11

530

我们使用的第二个机制是一个名为 **move** 的标准库函数，它定义在 utility 头文件中。目前，关于 move 我们需要了解两个关键点。首先，当 reallocate 在新内存中构造 string 时，它必须调用 move 来表示希望使用 string 的移动构造函数，原因我们将在 13.6.1 节（第 470 页）中解释。如果它漏掉了 move 调用，将会使用 string 的拷贝构造函数。其次，我们通常不为 move 提供一个 using 声明（参见 3.1 节，第 74 页），原因将在 18.2.3 节（第 706 页）中解释。当我们使用 move 时，直接调用 std::move 而不是 move。

reallocate 成员

了解了这些知识，现在就可以编写 reallocate 成员了。首先调用 allocate 分配新内存空间。我们每次重新分配内存时都会将 StrVec 的容量加倍。如果 StrVec 为空，我们将分配容纳一个元素的空间：

```
void StrVec::reallocate()
{
    // 我们将分配当前大小两倍的内存空间
    auto newcapacity = size() ? 2 * size() : 1;
    // 分配新内存
    auto newdata = alloc.allocate(newcapacity);
    // 将数据从旧内存移动到新内存
    auto dest = newdata;      // 指向新数组中下一个空闲位置
    auto elem = elements;     // 指向旧数组中下一个元素
    for (size_t i = 0; i != size(); ++i)
        alloc.construct(dest++, std::move(*elem++));
    free(); // 一旦我们移动完元素就释放旧内存空间
    // 更新我们的数据结构，执行新元素
    elements = newdata;
    first_free = dest;
    cap = elements + newcapacity;
}
```

for 循环遍历每个已有元素，并在新内存空间中 construct 一个对应元素。我们使用 dest 指向构造新 string 的内存，使用 elem 指向原数组中的元素。我们每次用后置递增运算将 dest（和 elem）推进到各自数组中的下一个元素。

construct 的第二个参数（即，确定使用哪个构造函数的参数（参见 12.2.2 节，第 428 页））是 move 返回的值。调用 move 返回的结果会令 construct 使用 string 的移

动构造函数。由于我们使用了移动构造函数，这些 string 管理的内存将不会被拷贝。相反，我们构造的每个 string 都会从 elem 指向的 string 那里接管内存的所有权。

531 在元素移动完毕后，我们调用 free 销毁旧元素并释放 StrVec 原来使用的内存。string 成员不再管理它们曾经指向的内存；其数据的管理职责已经转移给新 StrVec 内存中的元素了。我们不知道旧 StrVec 内存中的 string 包含什么值，但我们保证对它们执行 string 的析构函数是安全的。

剩下的就是更新指针，指向新分配并已初始化过的数组了。first_free 和 cap 指针分别被设置为指向最后一个构造的元素之后的位置及指向新分配空间的尾后位置。

13.5 节练习

练习 13.39：编写你自己版本的 StrVec，包括自己版本的 reserve、capacity（参见 9.4 节，第 318 页）和 resize（参见 9.3.5 节，第 314 页）。

练习 13.40：为你的 StrVec 类添加一个构造函数，它接受一个 initializer_list <string>参数。

练习 13.41：在 push_back 中，我们为什么在 construct 调用中使用前置递增运算？如果使用后置递增运算的话，会发生什么？

练习 13.42：在你的 TextQuery 和 QueryResult 类（参见 12.3 节，第 431 页）中用你的 StrVec 类代替 vector<string>，以此来测试你的 StrVec 类。

练习 13.43：重写 free 成员，用 for_each 和 lambda（参见 10.3.2 节，第 346 页）来代替 for 循环 destroy 元素。你更倾向于哪种实现，为什么？

练习 13.44：编写标准库 string 类的简化版本，命名为 String。你的类应该至少有一个默认构造函数和一个接受 C 风格字符串指针参数的构造函数。使用 allocator 为你的 String 类分配所需内存。

13.6 对象移动

新标准的一个最主要的特性是可以移动而非拷贝对象的能力。如我们在 13.1.1 节（第 440 页）中所见，很多情况下都会发生对象拷贝。在其中某些情况下，对象拷贝后就立即被销毁了。在这些情况下，移动而非拷贝对象会大幅度提升性能。

如我们已经看到的，我们的 StrVec 类是这种不必要的拷贝的一个很好的例子。在重新分配内存的过程中，从旧内存将元素拷贝到新内存是不必要的，更好的方式是移动元素。使用移动而不是拷贝的另一个原因源于 IO 类或 unique_ptr 这样的类。这些类都包含不能被共享的资源（如指针或 IO 缓冲）。因此，这些类型的对象不能拷贝但可以移动。

532 在旧 C++标准中，没有直接的方法移动对象。因此，即使不必拷贝对象的情况下，我们也不得不拷贝。如果对象较大，或者是对象本身要求分配内存空间（如 string），进行不必要的拷贝代价非常高。类似的，在旧版本的标准库中，容器中所保存的类必须是可拷贝的。但在新标准中，我们可以用容器保存不可拷贝的类型，只要它们能被移动即可。

标准库容器、string 和 shared_ptr 类既支持移动也支持拷贝。IO 类和 unique_ptr 类可以移动但不能拷贝。

13.6.1 右值引用

为了支持移动操作，新标准引入了一种新的引用类型——**右值引用**（rvalue reference）。所谓右值引用就是必须绑定到右值的引用。我们通过&&而不是&来获得右值引用。如我们将要看到的，右值引用有一个重要的性质——只能绑定到一个将要销毁的对象。因此，我们可以自由地将一个右值引用的资源“移动”到另一个对象中。

C++ 11

回忆一下，左值和右值是表达式的属性（参见 4.1.1 节，第 121 页）。一些表达式生成或要求左值，而另外一些则生成或要求右值。一般而言，一个左值表达式表示的是一个对象的身份，而一个右值表达式表示的是对象的值。

类似任何引用，一个右值引用也不过是某个对象的另一个名字而已。如我们所知，对于常规引用（为了与右值引用区分开来，我们可以称之为**左值引用**（lvalue reference）），我们不能将其绑定到要求转换的表达式、字面常量或是返回右值的表达式（参见 2.3.1 节，第 46 页）。右值引用有着完全相反的绑定特性：我们可以将一个右值引用绑定到这类表达式上，但不能将一个右值引用直接绑定到一个左值上：

```
int i = 42;
int &r = i;                // 正确：r 引用 i
int &&rr = i;              // 错误：不能将一个右值引用绑定到一个左值上
int &r2 = i * 42;          // 错误：i*42 是一个右值
const int &r3 = i * 42;    // 正确：我们可以将一个 const 的引用绑定到一个右值上
int &&rr2 = i * 42;        // 正确：将 rr2 绑定到乘法结果上
```

返回左值引用的函数，连同赋值、下标、解引用和前置递增/递减运算符，都是返回左值的表达式的例子。我们可以将一个左值引用绑定到这类表达式的结果上。

返回非引用类型的函数，连同算术、关系、位以及后置递增/递减运算符，都生成右值。我们不能将一个左值引用绑定到这类表达式上，但我们可以将一个 const 的左值引用或者一个右值引用绑定到这类表达式上。

左值持久：右值短暂

533

考察左值和右值表达式的列表，两者相互区别之处就很明显了：左值有持久的状态，而右值要么是字面常量，要么是在表达式求值过程中创建的临时对象。

由于右值引用只能绑定到临时对象，我们得知

- 所引用的对象将要被销毁
- 该对象没有其他用户

这两个特性意味着：使用右值引用的代码可以自由地接管所引用的对象的资源。

右值引用指向将要被销毁的对象。因此，我们可以从绑定到右值引用的对象“窃取”状态。

变量是左值

变量可以看作只有一个运算对象而没有运算符的表达式，虽然我们很少这样看待变

量。类似其他任何表达式，变量表达式也有左值/右值属性。变量表达式都是左值。带来的结果就是，我们不能将一个右值引用绑定到一个右值引用类型的变量上，这有些令人惊讶：

```
int &&rr1 = 42;  // 正确：字面常量是右值
int &&rr2 = rr1; // 错误：表达式 rr1 是左值!
```

其实有了右值表示临时对象这一观察结果，变量是左值这一特性并不令人惊讶。毕竟，变量是持久的，直至离开作用域时才被销毁。

变量是左值，因此我们不能将一个右值引用直接绑定到一个变量上，即使这个变量是右值引用类型也不行。

标准库 move 函数

虽然不能将一个右值引用直接绑定到一个左值上，但我们可以显式地将一个左值转换为对应的右值引用类型。我们还可以通过调用一个名为 **move** 的新标准库函数来获得绑定到左值上的右值引用，此函数定义在头文件 utility 中。move 函数使用了我们将在 16.2.6 节（第 610 页）中描述的机制来返回给定对象的右值引用。

C++ 11

```
int &&rr3 = std::move(rr1); // ok
```

move 调用告诉编译器：我们有一个左值，但我们希望像一个右值一样处理它。我们必须认识到，调用 move 就意味着承诺：除了对 rr1 赋值或销毁它外，我们将不再使用它。在调用 move 之后，我们不能对移后源对象的值做任何假设。

534

我们可以销毁一个移后源对象，也可以赋予它新值，但不能使用一个移后源对象的值。

如前所述，与大多数标准库名字的使用不同，对 move（参见 13.5 节，第 469 页）我们不提供 using 声明（参见 3.1 节，第 74 页）。我们直接调用 std::move 而不是 move，其原因将在 18.2.3 节（第 707 页）中解释。

使用 move 的代码应该使用 std::move 而不是 move。这样做可以避免潜在的名字冲突。

13.6.1 节练习

练习 13.45：解释右值引用和左值引用的区别。

练习 13.46：什么类型的引用可以绑定到下面的初始化器上？

```
int f();
vector<int> vi(100);
int? r1 = f();
int? r2 = vi[0];
int? r3 = r1;
int? r4 = vi[0] * f();
```

练习 13.47：对你在练习 13.44（13.5 节，第 470 页）中定义的 String 类，为它的拷贝构造函数和拷贝赋值运算符添加一条语句，在每次函数执行时打印一条信息。

练习 13.48：定义一个 vector<String>并在其上多次调用 push_back。运行你的程序，并观察 String 被拷贝了多少次。

13.6.2 移动构造函数和移动赋值运算符

类似 string 类（及其他标准库类），如果我们自己的类也同时支持移动和拷贝，那么也能从中受益。为了让我们自己的类型支持移动操作，需要为其定义移动构造函数和移动赋值运算符。这两个成员类似对应的拷贝操作，但它们从给定对象“窃取”资源而不是拷贝资源。

类似拷贝构造函数，移动构造函数的第一个参数是该类类型的一个引用。不同于拷贝构造函数的是，这个引用参数在移动构造函数中是一个右值引用。与拷贝构造函数一样，任何额外的参数都必须有默认实参。 C++ 11

除了完成资源移动，移动构造函数还必须确保移后源对象处于这样一个状态——销毁它是无害的。特别是，一旦资源完成移动，源对象必须不再指向被移动的资源——这些资源的所有权已经归属新创建的对象。

作为一个例子，我们为 StrVec 类定义移动构造函数，实现从一个 StrVec 到另一个 StrVec 的元素移动而非拷贝： 535

```
StrVec::StrVec(StrVec &&s) noexcept // 移动操作不应抛出任何异常
  // 成员初始化器接管 s 中的资源
  : elements(s.elements), first_free(s.first_free), cap(s.cap)
{
    // 令 s 进入这样的状态——对其运行析构函数是安全的
    s.elements = s.first_free = s.cap = nullptr;
}
```

我们将简短解释 noexcept（它通知标准库我们的构造函数不抛出任何异常），但让我们先分析一下此构造函数完成什么工作。

与拷贝构造函数不同，移动构造函数不分配任何新内存；它接管给定的 StrVec 中的内存。在接管内存之后，它将给定对象中的指针都置为 nullptr。这样就完成了从给定对象的移动操作，此对象将继续存在。最终，移后源对象会被销毁，意味着将在其上运行析构函数。StrVec 的析构函数在 first_free 上调用 deallocate。如果我们忘记了改变 s.first_free，则销毁移后源对象就会释放掉我们刚刚移动的内存。

移动操作、标准库容器和异常

由于移动操作“窃取”资源，它通常不分配任何资源。因此，移动操作通常不会抛出任何异常。当编写一个不抛出异常的移动操作时，我们应该将此事通知标准库。我们将看到，除非标准库知道我们的移动构造函数不会抛出异常，否则它会认为移动我们的类对象时可能会抛出异常，并且为了处理这种可能性而做一些额外的工作。

一种通知标准库的方法是在我们的构造函数中指明 noexcept。noexcept 是新标准引入的，我们将在 18.1.4 节（第 690 页）中讨论更多细节。目前重要的是要知道，noexcept 是我们承诺一个函数不抛出异常的一种方法。我们在一个函数的参数列表后指定 noexcept。在一个构造函数中，noexcept 出现在参数列表和初始化列表开始的冒号之间： C++ 11

```
class StrVec {
```

```
public:
    StrVec(StrVec&&) noexcept; // 移动构造函数
    // 其他成员的定义，如前
};
StrVec::StrVec(StrVec &&s) noexcept : /* 成员初始化器 */
{ /* 构造函数体 */ }
```

我们必须在类头文件的声明中和定义中（如果定义在类外的话）都指定 noexcept。

不抛出异常的移动构造函数和移动赋值运算符必须标记为 noexcept。

536

搞清楚为什么需要 noexcept 能帮助我们深入理解标准库是如何与我们自定义的类型交互的。我们需要指出一个移动操作不抛出异常，这是因为两个相互关联的事实：首先，虽然移动操作通常不抛出异常，但抛出异常也是允许的；其次，标准库容器能对异常发生时其自身的行为提供保障。例如，vector 保证，如果我们调用 push_back 时发生异常，vector 自身不会发生改变。

现在让我们思考 push_back 内部发生了什么。类似对应的 StrVec 操作（参见 13.5 节，第 466 页），对一个 vector 调用 push_back 可能要求为 vector 重新分配内存空间。当重新分配 vector 的内存时，vector 将元素从旧空间移动到新内存中，就像我们在 reallocate 中所做的那样（参见 13.5 节，第 469 页）。

如我们刚刚看到的那样，移动一个对象通常会改变它的值。如果重新分配过程使用了移动构造函数，且在移动了部分而不是全部元素后抛出了一个异常，就会产生问题。旧空间中的移动源元素已经被改变了，而新空间中未构造的元素可能尚不存在。在此情况下，vector 将不能满足自身保持不变的要求。

另一方面，如果 vector 使用了拷贝构造函数且发生了异常，它可以很容易地满足要求。在此情况下，当在新内存中构造元素时，旧元素保持不变。如果此时发生了异常，vector 可以释放新分配的（但还未成功构造的）内存并返回。vector 原有的元素仍然存在。

为了避免这种潜在问题，除非 vector 知道元素类型的移动构造函数不会抛出异常，否则在重新分配内存的过程中，它就必须使用拷贝构造函数而不是移动构造函数。如果希望在 vector 重新分配内存这类情况下对我们自定义类型的对象进行移动而不是拷贝，就必须显式地告诉标准库我们的移动构造函数可以安全使用。我们通过将移动构造函数（及移动赋值运算符）标记为 noexcept 来做到这一点。

移动赋值运算符

移动赋值运算符执行与析构函数和移动构造函数相同的工作。与移动构造函数一样，如果我们的移动赋值运算符不抛出任何异常，我们就应该将它标记为 noexcept。类似拷贝赋值运算符，移动赋值运算符必须正确处理自赋值：

```
StrVec &StrVec::operator=(StrVec &&rhs) noexcept
{
    // 直接检测自赋值
    if (this != &rhs) {
        free();                    // 释放已有元素
        elements = rhs.elements; // 从 rhs 接管资源
        first_free = rhs.first_free;
```

```
            cap = rhs.cap;
            // 将 rhs 置于可析构状态
            rhs.elements = rhs.first_free = rhs.cap = nullptr;
        }
        return *this;
    }
```

在此例中，我们直接检查 this 指针与 rhs 的地址是否相同。如果相同，右侧和左侧运算对象指向相同的对象，我们不需要做任何事情。否则，我们释放左侧运算对象所使用的内存，并接管给定对象的内存。与移动构造函数一样，我们将 rhs 中的指针置为 nullptr。 537

我们费心地去检查自赋值情况看起来有些奇怪。毕竟，移动赋值运算符需要右侧运算对象的一个右值。我们进行检查的原因是此右值可能是 move 调用的返回结果。与其他任何赋值运算符一样，关键点是我们不能在使用右侧运算对象的资源之前就释放左侧运算对象的资源（可能是相同的资源）。

移后源对象必须可析构

从一个对象移动数据并不会销毁此对象，但有时在移动操作完成后，源对象会被销毁。因此，当我们编写一个移动操作时，必须确保移后源对象进入一个可析构的状态。我们的 StrVec 的移动操作满足这一要求，这是通过将移后源对象的指针成员置为 nullptr 来实现的。

除了将移后源对象置为析构安全的状态之外，移动操作还必须保证对象仍然是有效的。一般来说，对象有效就是指可以安全地为其赋予新值或者可以安全地使用而不依赖其当前值。另一方面，移动操作对移后源对象中留下的值没有任何要求。因此，我们的程序不应该依赖于移后源对象中的数据。

例如，当我们从一个标准库 string 或容器对象移动数据时，我们知道移后源对象仍然保持有效。因此，我们可以对它执行诸如 empty 或 size 这些操作。但是，我们不知道将会得到什么结果。我们可能期望一个移后源对象是空的，但这并没有保证。

我们的 StrVec 类的移动操作将移后源对象置于与默认初始化的对象相同的状态。因此，我们可以继续对移后源对象执行所有的 StrVec 操作，与任何其他默认初始化的对象一样。而其他内部结构更为复杂的类，可能表现出完全不同的行为。

在移动操作之后，移后源对象必须保持有效的、可析构的状态，但是用户不能对其值进行任何假设。

合成的移动操作

与处理拷贝构造函数和拷贝赋值运算符一样，编译器也会合成移动构造函数和移动赋值运算符。但是，合成移动操作的条件与合成拷贝操作的条件大不相同。

回忆一下，如果我们不声明自己的拷贝构造函数或拷贝赋值运算符，编译器总会为我们合成这些操作（参见 13.1.1 节，第 440 页和 13.1.2 节，第 444 页）。拷贝操作要么被定义为逐成员拷贝，要么被定义为对象赋值，要么被定义为删除的函数。

与拷贝操作不同，编译器根本不会为某些类合成移动操作。特别是，如果一个类定义了自己的拷贝构造函数、拷贝赋值运算符或者析构函数，编译器就不会为它合成移动构造函数和移动赋值运算符了。因此，某些类就没有移动构造函数或移动赋值运算符。如我们将在第 477 页所见，如果一个类没有移动操作，通过正常的函数匹配，类会使用对应的拷 538

贝操作来代替移动操作。

只有当一个类没有定义任何自己版本的拷贝控制成员，且类的每个非 static 数据成员都可以移动时，编译器才会为它合成移动构造函数或移动赋值运算符。编译器可以移动内置类型的成员。如果一个成员是类类型，且该类有对应的移动操作，编译器也能移动这个成员：

```
// 编译器会为 X 和 hasX 合成移动操作
struct X {
    int i;              // 内置类型可以移动
    std::string s;      // string 定义了自己的移动操作
};
struct hasX {
    X mem; // X 有合成的移动操作
};
X x, x2 = std::move(x);           // 使用合成的移动构造函数
hasX hx, hx2 = std::move(hx);     // 使用合成的移动构造函数
```

只有当一个类没有定义任何自己版本的拷贝控制成员，且它的所有数据成员都能移动构造或移动赋值时，编译器才会为它合成移动构造函数或移动赋值运算符。

与拷贝操作不同，移动操作永远不会隐式定义为删除的函数。但是，如果我们显式地要求编译器生成=default 的（参见 7.1.4 节，第 237 页）移动操作，且编译器不能移动所有成员，则编译器会将移动操作定义为删除的函数。除了一个重要例外，什么时候将合成的移动操作定义为删除的函数遵循与定义删除的合成拷贝操作类似的原则（参见 13.1.6 节，第 449 页）：

- 与拷贝构造函数不同，移动构造函数被定义为删除的函数的条件是：有类成员定义了自己的拷贝构造函数且未定义移动构造函数，或者是有类成员未定义自己的拷贝构造函数且编译器不能为其合成移动构造函数。移动赋值运算符的情况类似。
- 如果有类成员的移动构造函数或移动赋值运算符被定义为删除的或是不可访问的，则类的移动构造函数或移动赋值运算符被定义为删除的。
- 类似拷贝构造函数，如果类的析构函数被定义为删除的或不可访问的，则类的移动构造函数被定义为删除的。
- 类似拷贝赋值运算符，如果有类成员是 const 的或是引用，则类的移动赋值运算符被定义为删除的。

539 例如，假定 Y 是一个类，它定义了自己的拷贝构造函数但未定义自己的移动构造函数：

```
// 假定 Y 是一个类，它定义了自己的拷贝构造函数但未定义自己的移动构造函数
struct hasY {
    hasY() = default;
    hasY(hasY&&) = default;
    Y mem; // hasY 将有一个删除的移动构造函数
};
hasY hy, hy2 = std::move(hy); // 错误：移动构造函数是删除的
```

编译器可以拷贝类型为 Y 的对象，但不能移动它们。类 hasY 显式地要求一个移动构造函数，但编译器无法为其生成。因此，hasY 会有一个删除的移动构造函数。如果 hasY 忽略了移动构造函数的声明，则编译器根本不能为它合成一个。如果移动操作可能被定义为

删除的函数，编译器就不会合成它们。

移动操作和合成的拷贝控制成员间还有最后一个相互作用关系：一个类是否定义了自己的移动操作对拷贝操作如何合成有影响。如果类定义了一个移动构造函数和/或一个移动赋值运算符，则该类的合成拷贝构造函数和拷贝赋值运算符会被定义为删除的。

定义了一个移动构造函数或移动赋值运算符的类必须也定义自己的拷贝操作。否则，这些成员默认地被定义为删除的。

移动右值，拷贝左值……

如果一个类既有移动构造函数，也有拷贝构造函数，编译器使用普通的函数匹配规则来确定使用哪个构造函数（参见 6.4 节，第 208 页）。赋值操作的情况类似。例如，在我们的 StrVec 类中，拷贝构造函数接受一个 const StrVec 的引用。因此，它可以用于任何可以转换为 StrVec 的类型。而移动构造函数接受一个 StrVec&&，因此只能用于实参是（非 static）右值的情形：

```
StrVec v1, v2;
v1 = v2;                        // v2 是左值；使用拷贝赋值
StrVec getVec(istream &);       // getVec 返回一个右值
v2 = getVec(cin);               // getVec(cin)是一个右值；使用移动赋值
```

在第一个赋值中，我们将 v2 传递给赋值运算符。v2 的类型是 StrVec，表达式 v2 是一个左值。因此移动版本的赋值运算符是不可行的（参见 6.6 节，第 217 页），因为我们不能隐式地将一个右值引用绑定到一个左值。因此，这个赋值语句使用拷贝赋值运算符。

在第二个赋值中，我们赋予 v2 的是 getVec 调用的结果。此表达式是一个右值。在此情况下，两个赋值运算符都是可行的——将 getVec 的结果绑定到两个运算符的参数都是允许的。调用拷贝赋值运算符需要进行一次到 const 的转换，而 StrVec&&则是精确匹配。因此，第二个赋值会使用移动赋值运算符。

……但如果没有移动构造函数，右值也被拷贝

540

如果一个类有一个拷贝构造函数但未定义移动构造函数，会发生什么呢？在此情况下，编译器不会合成移动构造函数，这意味着此类将有拷贝构造函数但不会有移动构造函数。如果一个类没有移动构造函数，函数匹配规则保证该类型的对象会被拷贝，即使我们试图通过调用 move 来移动它们时也是如此：

```
class Foo {
public:
    Foo() = default;
    Foo(const Foo&); // 拷贝构造函数
    // 其他成员定义，但 Foo 未定义移动构造函数
};
Foo x;
Foo y(x);              // 拷贝构造函数；x 是一个左值
Foo z(std::move(x)); // 拷贝构造函数，因为未定义移动构造函数
```

在对 z 进行初始化时，我们调用了 move(x)，它返回一个绑定到 x 的 Foo&&。Foo 的拷贝构造函数是可行的，因为我们可以将一个 Foo&&转换为一个 const Foo&。因此，z 的初始化将使用 Foo 的拷贝构造函数。

值得注意的是，用拷贝构造函数代替移动构造函数几乎肯定是安全的（赋值运算符的

情况类似)。一般情况下，拷贝构造函数满足对应的移动构造函数的要求：它会拷贝给定对象，并将原对象置于有效状态。实际上，拷贝构造函数甚至都不会改变原对象的值。

如果一个类有一个可用的拷贝构造函数而没有移动构造函数，则其对象是通过拷贝构造函数来“移动”的。拷贝赋值运算符和移动赋值运算符的情况类似。

拷贝并交换赋值运算符和移动操作

我们的 HasPtr 版本定义了一个拷贝并交换赋值运算符（参见 13.3 节，第 459 页），它是函数匹配和移动操作间相互关系的一个很好的示例。如果我们为此类添加一个移动构造函数，它实际上也会获得一个移动赋值运算符：

```
class HasPtr {
public:
    // 添加的移动构造函数
    HasPtr(HasPtr &&p) noexcept : ps(p.ps), i(p.i) {p.ps = 0;}
    // 赋值运算符既是移动赋值运算符，也是拷贝赋值运算符
    HasPtr& operator=(HasPtr rhs)
                    { swap(*this, rhs); return *this; }
    // 其他成员的定义，同 13.2.1 节（第 453 页）
};
```

在这个版本中，我们为类添加了一个移动构造函数，它接管了给定实参的值。构造函数体将给定的 HasPtr 的指针置为 0，从而确保销毁移后源对象是安全的。此函数不会抛出异常，因此我们将其标记为 noexcept（参见 13.6.2 节，第 473 页）。

541

现在让我们观察赋值运算符。此运算符有一个非引用参数，这意味着此参数要进行拷贝初始化（参见 13.1.1 节，第 441 页）。依赖于实参的类型，拷贝初始化要么使用拷贝构造函数，要么使用移动构造函数——左值被拷贝，右值被移动。因此，单一的赋值运算符就实现了拷贝赋值运算符和移动赋值运算符两种功能。

例如，假定 hp 和 hp2 都是 HasPtr 对象：

```
hp = hp2; // hp2 是一个左值；hp2 通过拷贝构造函数来拷贝
hp = std::move(hp2); // 移动构造函数移动 hp2
```

在第一个赋值中，右侧运算对象是一个左值，因此移动构造函数是不可行的。rhs 将使用拷贝构造函数来初始化。拷贝构造函数将分配一个新 string，并拷贝 hp2 指向的 string。

在第二个赋值中，我们调用 std:: move 将一个右值引用绑定到 hp2 上。在此情况下，拷贝构造函数和移动构造函数都是可行的。但是，由于实参是一个右值引用，移动构造函数是精确匹配的。移动构造函数从 hp2 拷贝指针，而不会分配任何内存。

不管使用的是拷贝构造函数还是移动构造函数，赋值运算符的函数体都 swap 两个运算对象的状态。交换 HasPtr 会交换两个对象的指针（及 int）成员。在 swap 之后，rhs 中的指针将指向原来左侧运算对象所拥有的 string。当 rhs 离开其作用域时，这个 string 将被销毁。

建议：更新三/五法则

所有五个拷贝控制成员应该看作一个整体：一般来说，如果一个类定义了任何一个

拷贝操作，它就应该定义所有五个操作。如前所述，某些类必须定义拷贝构造函数、拷贝赋值运算符和析构函数才能正确工作（参见 13.1.4 节，第 447 页）。这些类通常拥有一个资源，而拷贝成员必须拷贝此资源。一般来说，拷贝一个资源会导致一些额外开销。在这种拷贝并非必要的情况下，定义了移动构造函数和移动赋值运算符的类就可以避免此问题。

Message 类的移动操作

定义了自己的拷贝构造函数和拷贝赋值运算符的类通常也会从移动操作受益。例如，我们的 Message 和 Folder 类（参见 13.4 节，第 460 页）就应该定义移动操作。通过定义移动操作，Message 类可以使用 string 和 set 的移动操作来避免拷贝 contents 和 folders 成员的额外开销。

但是，除了移动 folders 成员，我们还必须更新每个指向原 Message 的 Folder。我们必须删除指向旧 Message 的指针，并添加一个指向新 Message 的指针。

移动构造函数和移动赋值运算符都需要更新 Folder 指针，因此我们首先定义一个操作来完成这一共同的工作：

542

```
// 从本 Message 移动 Folder 指针
void Message::move_Folders(Message *m)
{
    folders = std::move(m->folders); // 使用 set 的移动赋值运算符
    for (auto f : folders) { // 对每个 Folder
        f->remMsg(m);        // 从 Folder 中删除旧 Message
        f->addMsg(this);     // 将本 Message 添加到 Folder 中
    }
    m->folders.clear();      // 确保销毁 m 是无害的
}
```

此函数首先移动 folders 集合。通过调用 move，我们使用了 set 的移动赋值运算符而不是它的拷贝赋值运算符。如果我们忽略了 move 调用，代码仍能正常工作，但带来了不必要的拷贝。函数然后遍历所有 Folder，从其中删除指向原 Message 的指针并添加指向新 Message 的指针。

值得注意的是，向 set 插入一个元素可能会抛出一个异常——向容器添加元素的操作要求分配内存，意味着可能会抛出一个 bad_alloc 异常（参见 12.1.2 节，第 409 页）。因此，与我们的 HasPtr 和 StrVec 类的移动操作不同，Message 的移动构造函数和移动赋值运算符可能会抛出异常。因此我们未将它们标记为 noexcept（参见 13.6.2 节，第 473 页）。

函数最后对 m.folders 调用 clear。在执行了 move 之后，我们知道 m.folders 是有效的，但不知道它包含什么内容。由于 Message 的析构函数遍历 folders，我们希望能确定 set 是空的。

Message 的移动构造函数调用 move 来移动 contents，并默认初始化自己的 folders 成员：

```
Message::Message(Message &&m): contents(std::move(m.contents))
{
    move_Folders(&m); // 移动 folders 并更新 Folder 指针
}
```

在构造函数体中，我们调用了 move_Folders 来删除指向 m 的指针并插入指向本 Message 的指针。

移动赋值运算符直接检查自赋值情况：

```
Message& Message::operator=(Message &&rhs)
{
    if (this != &rhs) {        // 直接检查自赋值情况
        remove_from_Folders();
        contents = std::move(rhs.contents); // 移动赋值运算符
        move_Folders(&rhs);  // 重置 Folders 指向本 Message
    }
    return *this;
}
```

543 与任何赋值运算符一样，移动赋值运算符必须销毁左侧运算对象的旧状态。在本例中，销毁左侧运算对象要求我们从现有 folders 中删除指向本 Message 的指针，我们调用 remove_from_Folders 来完成这一工作。完成删除工作后，我们调用 move 从 rhs 将 contents 移动到 this 对象。剩下的就是调用 move_Messages 来更新 Folder 指针了。

移动迭代器

StrVec 的 reallocate 成员（参见 13.5 节，第 469 页）使用了一个 for 循环来调用 construct 从旧内存将元素拷贝到新内存中。作为一种替换方法，如果我们能调用 uninitialized_copy 来构造新分配的内存，将比循环更为简单。但是，uninitialized_copy 恰如其名：它对元素进行拷贝操作。标准库中并没有类似的函数将对象“移动”到未构造的内存中。

C++ 11 新标准库中定义了一种**移动迭代器**（move iterator）适配器（参见 10.4 节，第 358 页）。一个移动迭代器通过改变给定迭代器的解引用运算符的行为来适配此迭代器。一般来说，一个迭代器的解引用运算符返回一个指向元素的左值。与其他迭代器不同，移动迭代器的解引用运算符生成一个右值引用。

我们通过调用标准库的 make_move_iterator 函数将一个普通迭代器转换为一个移动迭代器。此函数接受一个迭代器参数，返回一个移动迭代器。

原迭代器的所有其他操作在移动迭代器中都照常工作。由于移动迭代器支持正常的迭代器操作，我们可以将一对移动迭代器传递给算法。特别是，可以将移动迭代器传递给 uninitialized_copy：

```
void StrVec::reallocate()
{
    // 分配大小两倍于当前规模的内存空间
    auto newcapacity = size() ? 2 * size() : 1;
    auto first = alloc.allocate(newcapacity);
    // 移动元素
    auto last = uninitialized_copy(make_move_iterator(begin()),
                                   make_move_iterator(end()),
                                   first);
    free();                // 释放旧空间
    elements = first;      // 更新指针
    first_free = last;
```

```
        cap = elements + newcapacity;
    }
```

uninitialized_copy 对输入序列中的每个元素调用 construct 来将元素“拷贝”到目的位置。此算法使用迭代器的解引用运算符从输入序列中提取元素。由于我们传递给它的是移动迭代器，因此解引用运算符生成的是一个右值引用，这意味着 construct 将使用移动构造函数来构造元素。

值得注意的是，标准库不保证哪些算法适用移动迭代器，哪些不适用。由于移动一个对象可能销毁掉原对象，因此你只有在确信算法在为一个元素赋值或将其传递给一个用户定义的函数后不再访问它时，才能将移动迭代器传递给算法。 544

建议：不要随意使用移动操作

由于一个移后源对象具有不确定的状态，对其调用 std::move 是危险的。当我们调用 move 时，必须绝对确认移后源对象没有其他用户。

通过在类代码中小心地使用 move，可以大幅度提升性能。而如果随意在普通用户代码（与类实现代码相对）中使用移动操作，很可能导致莫名其妙的、难以查找的错误，而难以提升应用程序性能。

Best Practices 在移动构造函数和移动赋值运算符这些类实现代码之外的地方，只有当你确信需要进行移动操作且移动操作是安全的，才可以使用 std:: move。

13.6.2 节练习

练习 13.49：为你的 StrVec、String 和 Message 类添加一个移动构造函数和一个移动赋值运算符。

练习 13.50：在你的 String 类的移动操作中添加打印语句，并重新运行 13.6.1 节（第 473 页）的练习 13.48 中的程序，它使用了一个 vector<String>，观察什么时候会避免拷贝。

练习 13.51：虽然 unique_ptr 不能拷贝，但我们在 12.1.5 节（第 418 页）中编写了一个 clone 函数，它以值方式返回一个 unique_ptr。解释为什么函数是合法的，以及为什么它能正确工作。

练习 13.52：详细解释第 478 页中的 HasPtr 对象的赋值发生了什么？特别是，一步一步描述 hp、hp2 以及 HasPtr 的赋值运算符中的参数 rhs 的值发生了什么变化。

练习 13.53：从底层效率的角度看，HasPtr 的赋值运算符并不理想，解释为什么。为 HasPtr 实现一个拷贝赋值运算符和一个移动赋值运算符，并比较你的新的移动赋值运算符中执行的操作和拷贝并交换版本中执行的操作。

练习 13.54：如果我们为 HasPtr 定义了移动赋值运算符，但未改变拷贝并交换运算符，会发生什么？编写代码验证你的答案。

13.6.3 右值引用和成员函数

除了构造函数和赋值运算符之外，如果一个成员函数同时提供拷贝和移动版本，它也能从中受益。这种允许移动的成员函数通常使用与拷贝/移动构造函数和赋值运算符相同的参数模式——一个版本接受一个指向 const 的左值引用，第二个版本接受一个指向非 545

const 的右值引用。

例如，定义了 push_back 的标准库容器提供两个版本：一个版本有一个右值引用参数，而另一个版本有一个 const 左值引用。假定 X 是元素类型，那么这些容器就会定义以下两个 push_back 版本：

```
void push_back(const X&);      // 拷贝：绑定到任意类型的 X
void push_back(X&&);           // 移动：只能绑定到类型 X 的可修改的右值
```

我们可以将能转换为类型 X 的任何对象传递给第一个版本的 push_back。此版本从其参数拷贝数据。对于第二个版本，我们只可以传递给它非 const 的右值。此版本对于非 const 的右值是精确匹配（也是更好的匹配）的，因此当我们传递一个可修改的右值（参见 13.6.2 节，第 477 页）时，编译器会选择运行这个版本。此版本会从其参数窃取数据。

一般来说，我们不需要为函数操作定义接受一个 const X&&或是一个（普通的）X&参数的版本。当我们希望从实参“窃取”数据时，通常传递一个右值引用。为了达到这一目的，实参不能是 const 的。类似的，从一个对象进行拷贝的操作不应该改变该对象。因此，通常不需要定义一个接受一个（普通的）X&参数的版本。

区分移动和拷贝的重载函数通常有一个版本接受一个 const T&，而另一个版本接受一个 T&&。

作为一个更具体的例子，我们将为 StrVec 类定义另一个版本的 push_back：

```
class StrVec {
public:
    void push_back(const std::string&);   // 拷贝元素
    void push_back(std::string&&);        // 移动元素
    // 其他成员的定义，如前
};
// 与 13.5 节（第 466 页）中的原版本相同
void StrVec::push_back(const string& s)
{
    chk_n_alloc(); // 确保有空间容纳新元素
    // 在 first_free 指向的元素中构造 s 的一个副本
    alloc.construct(first_free++, s);
}
void StrVec::push_back(string &&s)
{
    chk_n_alloc(); // 如果需要的话为 StrVec 重新分配内存
    alloc.construct(first_free++, std::move(s));
}
```

这两个成员几乎是相同的。差别在于右值引用版本调用 move 来将其参数传递给 construct。如前所述，construct 函数使用其第二个和随后的实参的类型来确定使用哪个构造函数。由于 move 返回一个右值引用，传递给 construct 的实参类型是 string&&。因此，会使用 string 的移动构造函数来构造新元素。

546

当我们调用 push_back 时，实参类型决定了新元素是拷贝还是移动到容器中：

```
StrVec vec; // 空 StrVec
string s = "some string or another";
vec.push_back(s);          // 调用 push_back(const string&)
```

```
vec.push_back("done");    // 调用 push_back(string&&)
```

这些调用的差别在于实参是一个左值还是一个右值（从"done"创建的临时 string），具体调用哪个版本据此来决定。

右值和左值引用成员函数

通常，我们在一个对象上调用成员函数，而不管该对象是一个左值还是一个右值。例如：

```
string s1 = "a value", s2 = "another";
auto n = (s1 + s2).find('a');
```

此例中，我们在一个 string 右值上调用 find 成员（参见 9.5.3 节，第 325 页），该 string 右值是通过连接两个 string 而得到的。有时，右值的使用方式可能令人惊讶：

```
s1 + s2 = "wow!";
```

此处我们对两个 string 的连接结果——一个右值，进行了赋值。

在旧标准中，我们没有办法阻止这种使用方式。为了维持向后兼容性，新标准库类仍然允许向右值赋值。但是，我们可能希望在自己的类中阻止这种用法。在此情况下，我们希望强制左侧运算对象（即，this 指向的对象）是一个左值。

我们指出 this 的左值/右值属性的方式与定义 const 成员函数相同（参见 7.1.2 节，第 231 页），即，在参数列表后放置一个**引用限定符**（reference qualifier）：

C++ 11

```
class Foo {
public:
    Foo &operator=(const Foo&) &; // 只能向可修改的左值赋值
    // Foo 的其他参数
};
Foo &Foo::operator=(const Foo &rhs) &
{
    // 执行将 rhs 赋予本对象所需的工作
    return *this;
}
```

引用限定符可以是&或&&，分别指出 this 可以指向一个左值或右值。类似 const 限定符，引用限定符只能用于（非 static）成员函数，且必须同时出现在函数的声明和定义中。

对于&限定的函数，我们只能将它用于左值；对于&&限定的函数，只能用于右值：

547

```
Foo &retFoo();    // 返回一个引用；retFoo 调用是一个左值
Foo retVal();     // 返回一个值；retVal 调用是一个右值
Foo i, j;         // i 和 j 是左值
i = j;            // 正确：i 是左值
retFoo() = j;     // 正确：retFoo()返回一个左值
retVal() = j;     // 错误：retVal()返回一个右值
i = retVal();     // 正确：我们可以将一个右值作为赋值操作的右侧运算对象
```

一个函数可以同时用 const 和引用限定。在此情况下，引用限定符必须跟随在 const 限定符之后：

```
class Foo {
public:
Foo someMem() & const;        // 错误：const 限定符必须在前
Foo anotherMem() const &;     // 正确：const 限定符在前
};
```

重载和引用函数

就像一个成员函数可以根据是否有 const 来区分其重载版本一样（参见 7.3.2 节，第 247 页），引用限定符也可以区分重载版本。而且，我们可以综合引用限定符和 const 来区分一个成员函数的重载版本。例如，我们将为 Foo 定义一个名为 data 的 vector 成员和一个名为 sorted 的成员函数，sorted 返回一个 Foo 对象的副本，其中 vector 已被排序：

```
class Foo {
public:
    Foo sorted() &&;          // 可用于可改变的右值
    Foo sorted() const &;     // 可用于任何类型的 Foo
    // Foo 的其他成员的定义
private:
    vector<int> data;
};
// 本对象为右值，因此可以原址排序
Foo Foo::sorted() &&
{
    sort(data.begin(), data.end());
    return *this;
}
// 本对象是 const 或是一个左值，哪种情况我们都不能对其进行原址排序
Foo Foo::sorted() const & {
    Foo ret(*this);                          // 拷贝一个副本
    sort(ret.data.begin(), ret.data.end());  // 排序副本
    return ret;                              // 返回副本
}
```

当我们对一个右值执行 sorted 时，它可以安全地直接对 data 成员进行排序。对象是一个右值，意味着没有其他用户，因此我们可以改变对象。当对一个 const 右值或一个左值执行 sorted 时，我们不能改变对象，因此就需要在排序前拷贝 data。

编译器会根据调用 sorted 的对象的左值/右值属性来确定使用哪个 sorted 版本：

548

```
retVal().sorted(); // retVal()是一个右值，调用 Foo::sorted() &&
retFoo().sorted(); // retFoo()是一个左值，调用 Foo::sorted() const &
```

当我们定义 const 成员函数时，可以定义两个版本，唯一的差别是一个版本有 const 限定而另一个没有。引用限定的函数则不一样。如果我们定义两个或两个以上具有相同名字和相同参数列表的成员函数，就必须对所有函数都加上引用限定符，或者所有都不加：

```
class Foo {
public:
    Foo sorted() &&;
    Foo sorted() const; // 错误：必须加上引用限定符
    // Comp 是函数类型的类型别名（参见 6.7 节，第 222 页）
    // 此函数类型可以用来比较 int 值
    using Comp = bool(const int&, const int&);
    Foo sorted(Comp*);            // 正确：不同的参数列表
    Foo sorted(Comp*) const;      // 正确：两个版本都没有引用限定符
    };
```

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。

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。

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。

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。 554

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：我们能把一个 const char*加到一个 string 对象中
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。

557

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，第二个形参是将要读入到的（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。

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。

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 &lhs, const Sales_data &rhs)
{
    Sales_data sum = lhs;          // 把 lhs 的数据成员拷贝给 sum
    sum += rhs;                    // 将 rhs 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 lhs 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 rhs 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 &lhs, const Sales_data &rhs)
{
    return lhs.isbn() == rhs.isbn() &&
           lhs.units_sold == rhs.units_sold &&
           lhs.revenue == rhs.revenue;
}
bool operator!=(const Sales_data &lhs, const Sales_data &rhs)
```

```
{
    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。
- 562 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==时肯定也希望能使用!=，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。

如果某个类在逻辑上有相等性的含义，则该类应该定义 operator==，这样做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。

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

复合赋值运算符不非得是类的成员，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。

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：

566

```
// 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
}
```

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。

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
};
```

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
};
```

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。

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号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：

573

```
// 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：

574

```
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运算对象执行+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
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
```

575

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。当我们需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	
function<T> f;	f 是一个用来存储可调用对象的空 function，这些可调用对象的调用形式应该与函数类型 T 相同（即 T 是 *retType(args)*）
function<T> f(nullptr);	显式地构造一个空 function
function<T> f(obj);	在 f 中存储可调用对象 obj 的副本
f	将 f 作为条件：当 f 含有一个可调用对象时为真；否则为假
f(*args*)	调用 f 中的对象，参数是 *args*
定义为 function<T>的成员的类型	
result_type	该 function 类型的可调用对象返回的类型
argument_type first_argument_type second_argument_type	当 T 有一个或两个实参时定义的类型。如果 T 只有一个实参，则 argument_type 是该类型的同义词；如果 T 有两个实参，则 first_argument_type 和 second_argument_type 分别代表两个实参的类型

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：

579

```
// 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）。

580

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：

581

```
// 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。

582

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

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会具有二义性。

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，也不要在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。

当我们试图用 long 初始化 a2 时也遇到了同样问题，哪个构造函数都无法精确匹配 long 类型。它们在使用构造函数前都要求先将实参进行类型转换：

- 先执行 long 到 double 的标准类型转换，再执行 A(double)
- 先执行 long 到 int 的标准类型转换，再执行 A(int)

编译器没办法区分这两种转换序列的好坏，因此该调用将产生二义性。

调用 f2 及初始化 a2 的过程之所以会产生二义性，根本原因是它们所需的标准类型转换级别一致（参见 6.6.1 节，第 219 页）。当我们使用用户定义的类型转换时，如果转换过程包含标准类型转换，则标准类型转换的级别将决定编译器选择最佳匹配的过程：

```
short s = 42;
// 把 short 提升成 int 优于把 short 转换成 double
A a3(s);                    // 使用 A::A(int)
```

在此例中，把 short 提升成 int 的操作要优于把 short 转换成 double 的操作，因此编译器将使用 A::A(int) 构造函数构造 a3，其中实参是 s（提升后）的值。

当我们使用两个用户定义的类型转换时，如果转换函数之前或之后存在标准类型转换，则标准类型转换将决定最佳匹配到底是哪个。

提示：类型转换与运算符

586

要想正确地设计类的重载运算符、转换构造函数及类型转换函数，必须加倍小心。尤其是当类同时定义了类型转换运算符及重载运算符时特别容易产生二义性。以下的经验规则可能对你有所帮助：

- 不要令两个类执行相同的类型转换：如果 Foo 类有一个接受 Bar 类对象的构造函数，则不要在 Bar 类中再定义转换目标是 Foo 类的类型转换运算符。
- 避免转换目标是内置算术类型的类型转换。特别是当你已经定义了一个转换成算术类型的类型转换时，接下来
  - 不要再定义接受算术类型的重载运算符。如果用户需要使用这样的运算符，则类型转换操作将转换你的类型的对象，然后使用内置的运算符。
  - 不要定义转换到多种算术类型的类型转换。让标准类型转换完成向其他算术类型转换的工作。

一言以蔽之：除了显式地向 bool 类型的转换之外，我们应该尽量避免定义类型转换函数并尽可能地限制那些“显然正确”的非显式构造函数。

重载函数与转换构造函数

当我们调用重载的函数时，从多个类型转换中进行选择将变得更加复杂。如果两个或多个类型转换都提供了同一种可行匹配，则这些类型转换一样好。

举个例子，当几个重载函数的参数分属不同的类类型时，如果这些类恰好定义了同样的转换构造函数，则二义性问题将进一步提升：

```
struct C {
    C(int);
    // 其他成员
```

```
};
struct D {
    D(int);
    // 其他成员
};
void manip(const C&);
void manip(const D&);
manip(10);          // 二义性错误：含义是 manip(C(10))还是 manip(D(10))
```

其中 C 和 D 都包含接受 int 的构造函数，两个构造函数各自匹配 manip 的一个版本。因此调用将具有二义性：它的含义可能是把 int 转换成 C，然后调用 manip 的第一个版本；也可能是把 int 转换成 D，然后调用 manip 的第二个版本。

调用者可以显式地构造正确的类型从而消除二义性：

```
manip(C(10));     // 正确：调用 manip(const C&)
```

如果在调用重载函数时我们需要使用构造函数或者强制类型转换来改变实参的类型，则这通常意味着程序的设计存在不足。

重载函数与用户定义的类型转换

当调用重载函数时，如果两个（或多个）用户定义的类型转换都提供了可行匹配，则我们认为这些类型转换一样好。在这个过程中，我们不会考虑任何可能出现的标准类型转换的级别。只有当重载函数能通过同一个类型转换函数得到匹配时，我们才会考虑其中出现的标准类型转换。

587

例如当我们调用 manip 时，即使其中一个类定义了需要对实参进行标准类型转换的构造函数，这次调用仍然会具有二义性：

```
struct E {
    E(double);
    // 其他成员
};
void manip2(const C&);
void manip2(const E&);
// 二义性错误：两个不同的用户定义的类型转换都能用在此处
manip2(10);       // 含义是 manip2(C(10))还是 manip2(E(double(10)))
```

在此例中，C 有一个转换源为 int 的类型转换，E 有一个转换源为 double 的类型转换。对于 manip2(10)来说，两个 manip2 函数都是可行的：

- manip2(const C&)是可行的，因为 C 有一个接受 int 的转换构造函数，该构造函数与实参精确匹配。
- manip2(const E&)是可行的，因为 E 有一个接受 double 的转换构造函数，而且为了使用该函数我们可以利用标准类型转换把 int 转换成所需的类型。

因为调用重载函数所请求的用户定义的类型转换不止一个且彼此不同，所以该调用具有二义性。即使其中一个调用需要额外的标准类型转换而另一个调用能精确匹配，编译器也会将该调用标示为错误。

在调用重载函数时，如果需要额外的标准类型转换，则该转换的级别只有当所有可行函数都请求同一个用户定义的类型转换时才有用。如果所需的用户定义的类型转换不止一个，则该调用具有二义性。

14.9.2 节练习

练习 14.50：在初始化 ex1 和 ex2 的过程中，可能用到哪些类类型的转换序列呢？说明初始化是否正确并解释原因。

```
struct LongDouble {
    LongDouble(double = 0.0);
    operator double();
    operator float();
};
LongDouble ldObj;
int ex1 = ldObj;
float ex2 = ldObj;
```

练习 14.51：在调用 calc 的过程中，可能用到哪些类型转换序列呢？说明最佳可行函数是如何被选出来的。

```
void calc(int);
void calc(LongDouble);
double dval;
calc(dval);                    // 哪个 calc?
```

14.9.3 函数匹配与重载运算符

重载的运算符也是重载的函数。因此，通用的函数匹配规则（参见 6.4 节，第 208 页）同样适用于判断在给定的表达式中到底应该使用内置运算符还是重载的运算符。不过当运算符函数出现在表达式中时，候选函数集的规模要比我们使用调用运算符调用函数时更大。如果 a 是一种类类型，则表达式 a *sym* b 可能是

```
a.operatorsym(b); // a 有一个 operatorsym 成员函数
operatorsym(a, b);// operatorsym 是一个普通函数
```

和普通函数调用不同，我们不能通过调用的形式来区分当前调用的是成员函数还是非成员函数。

当我们使用重载运算符作用于类类型的运算对象时，候选函数中包含该运算符的普通非成员版本和内置版本。除此之外，如果左侧运算对象是类类型，则定义在该类中的运算符的重载版本也包含在候选函数内。 588

当我们调用一个命名的函数时，具有该名字的成员函数和非成员函数不会彼此重载，这是因为我们用来调用命名函数的语法形式对于成员函数和非成员函数来说是不相同的。当我们通过类类型的对象（或者该对象的指针及引用）进行函数调用时，只考虑该类的成员函数。而当我们在表达式中使用重载的运算符时，无法判断正在使用的是成员函数还是非成员函数，因此二者都应该在考虑的范围内。

表达式中运算符的候选函数集既应该包括成员函数，也应该包括非成员函数。

举个例子，我们为 SmallInt 类定义一个加法运算符：

```
class SmallInt {
    friend
    SmallInt operator+(const SmallInt&, const SmallInt&);
public:
    SmallInt(int = 0);                          // 转换源为 int 的类型转换
    operator int() const { return val; }  // 转换目标为 int 的类型转换
private:
    std::size_t val;
};
```

589 可以使用这个类将两个 SmallInt 对象相加，但如果我们试图执行混合模式的算术运算，就将遇到二义性的问题：

```
SmallInt s1, s2;
SmallInt s3 = s1 + s2;        // 使用重载的 operator+
int i = s3 + 0;               // 二义性错误
```

第一条加法语句接受两个 SmallInt 值并执行+运算符的重载版本。第二条加法语句具有二义性：因为我们可以把 0 转换成 SmallInt，然后使用 SmallInt 的+；或者把 s3 转换成 int，然后对于两个 int 执行内置的加法运算。

如果我们对同一个类既提供了转换目标是算术类型的类型转换，也提供了重载的运算符，则将会遇到重载运算符与内置运算符的二义性问题。

14.9.3 节练习

练习 14.52：在下面的加法表达式中分别选用了哪个 operator+？列出候选函数、可行函数及为每个可行函数的实参执行的类型转换：

```
struct LongDouble {
    // 用于演示的成员 operator+；在通常情况下+是个非成员
    LongDouble operator+(const SmallInt&);
    // 其他成员与 14.9.2 节（第 521 页）一致
};
LongDouble operator+(LongDouble&, double);
SmallInt si;
LongDouble ld;
ld = si + ld;
ld = ld + si;
```

练习 14.53：假设我们已经定义了如第 522 页所示的 SmallInt，判断下面的加法表达式是否合法。如果合法，使用了哪个加法运算符？如果不合法，应该怎样修改代码才能使其合法？

```
SmallInt s1;
double d = s1 + 3.14;
```

小结

590

一个重载的运算符必须是某个类的成员或者至少拥有一个类类型的运算对象。重载运算符的运算对象数量、结合律、优先级与对应的用于内置类型的运算符完全一致。当运算符被定义为类的成员时，类对象的隐式 this 指针绑定到第一个运算对象。赋值、下标、函数调用和箭头运算符必须作为类的成员。

如果类重载了函数调用运算符 operator()，则该类的对象被称作“函数对象”。这样的对象常用在标准函数中。lambda 表达式是一种简便的定义函数对象类的方式。

在类中可以定义转换源或转换目的是该类型本身的类型转换，这样的类型转换将自动执行。只接受单独一个实参的非显式构造函数定义了从实参类型到类类型的类型转换；而非显式的类型转换运算符则定义了从类类型到其他类型的转换。

术语表

调用形式（call signature） 表示一个可调用对象的接口。在调用形式中包括返回类型以及一个实参类型列表，该列表在一对圆括号内，实参类型之间以逗号分隔。

类类型转换（class-type conversion） 包括由构造函数定义的从其他类型到类类型的转换以及由类型转换运算符定义的从类类型到其他类型的转换。只接受单独一个实参的非显式构造函数定义了从实参类型到类类型的转换；而类型转换运算符则定义了从类类型到某个指定类型的转换。

类型转换运算符（conversion operator） 是类的成员函数，定义了从类类型到其他类型的转换。类型转换运算符必须是它要转换的类的成员，并且通常被定义为常量成员。这类运算符既没有返回类型，也不接受参数。它们返回一个可变为转换运算符类型的值，也就是说，operator int 返回一个 int，operator string 返回一个 string，依此类推。

显式的类型转换运算符（explicit conversion operator） 由关键字 explicit 限定的类型转换运算符。这样的运算符用于条件中的隐式类型转换。

函数对象（function object） 定义了重载调用运算符的对象。在需要使用函数的地方都能使用函数对象。

函数表（function table） 形如 map 或 vector 的容器，容器中所存的值可以被调用。

函数模板（function template） 能够表示任意可调用类型的标准库模板。

重载的运算符（overloaded operator） 重定义了某种内置运算符的含义的函数。重载的运算符函数含有关键字 operator，之后是要定义的符号。重载的运算符必须含有至少一个类类型的运算对象。重载运算符的优先级、结合律、运算对象数量都与其内置版本一致。

用户定义的类型转换（user-defined conversion） 类类型转换的同义词。

第 15 章
面向对象程序设计

内容

15.1 OOP：概述 526
15.2 定义基类和派生类 527
15.3 虚函数 536
15.4 抽象基类 540
15.5 访问控制与继承 542
15.6 继承中的类作用域 547
15.7 构造函数与拷贝控制 551
15.8 容器与继承 558
15.9 文本查询程序再探 562
小结 575
术语表 575

面向对象程序设计基于三个基本概念：数据抽象、继承和动态绑定。第 7 章已经介绍了数据抽象的知识，本章将介绍继承和动态绑定。

继承和动态绑定对程序的编写有两方面的影响：一是我们可以更容易地定义与其他类相似但不完全相同的新类；二是在使用这些彼此相似的类编写程序时，我们可以在一定程度上忽略掉它们的区别。

592 在很多程序中都存在着一些相互关联但是有细微差别的概念。例如，书店中不同书籍的定价策略可能不同：有的书籍按原价销售，有的则打折销售。有时，我们给那些购买书籍超过一定数量的顾客打折；另一些时候，则只对前多少本销售的书籍打折，之后就调回原价，等等。面向对象的程序设计（OOP）适用于这类应用。

15.1 OOP：概述

面向对象程序设计（object-oriented programming）的核心思想是数据抽象、继承和动态绑定。通过使用数据抽象，我们可以将类的接口与实现分离（见第 7 章）；使用继承，可以定义相似的类型并对其相似关系建模；使用动态绑定，可以在一定程度上忽略相似类型的区别，而以统一的方式使用它们的对象。

继承

通过**继承**（inheritance）联系在一起的类构成一种层次关系。通常在层次关系的根部有一个**基类**（base class），其他类则直接或间接地从基类继承而来，这些继承得到的类称为**派生类**（derived class）。基类负责定义在层次关系中所有类共同拥有的成员，而每个派生类定义各自特有的成员。

为了对之前提到的不同定价策略建模，我们首先定义一个名为 Quote 的类，并将它作为层次关系中的基类。Quote 的对象表示按原价销售的书籍。Quote 派生出另一个名为 Bulk_quote 的类，它表示可以打折销售的书籍。

这些类将包含下面的两个成员函数：

- isbn()，返回书籍的 ISBN 编号。该操作不涉及派生类的特殊性，因此只定义在 Quote 类中。
- net_price(size_t)，返回书籍的实际销售价格，前提是用户购买该书的数量达到一定标准。这个操作显然是类型相关的，Quote 和 Bulk_quote 都应该包含该函数。

在 C++语言中，基类将类型相关的函数与派生类不做改变直接继承的函数区分对待。对于某些函数，基类希望它的派生类各自定义适合自身的版本，此时基类就将这些函数声明成**虚函数**（virtual function）。因此，我们可以将 Quote 类编写成：

```
class Quote {
public:
    std::string isbn() const;
    virtual double net_price(std::size_t n) const;
};
```

593 派生类必须通过使用**类派生列表**（class derivation list）明确指出它是从哪个（哪些）基类继承而来的。类派生列表的形式是：首先是一个冒号，后面紧跟以逗号分隔的基类列表，其中每个基类前面可以有访问说明符：

```
class Bulk_quote : public Quote {          // Bulk_quote 继承了 Quote
public:
     double net_price(std::size_t) const override;
};
```

因为 Bulk_quote 在它的派生列表中使用了 public 关键字，因此我们完全可以把

Bulk_quote 的对象当成 Quote 的对象来使用。

派生类必须在其内部对所有重新定义的虚函数进行声明。派生类可以在这样的函数之前加上 virtual 关键字，但是并不是非得这么做。出于 15.3 节（第 538 页）将要解释的原因，C++11 新标准允许派生类显式地注明它将使用哪个成员函数改写基类的虚函数，具体措施是在该函数的形参列表之后增加一个 override 关键字。

动态绑定

通过使用**动态绑定**（dynamic binding），我们能用同一段代码分别处理 Quote 和 Bulk_quote 的对象。例如，当要购买的书籍和购买的数量都已知时，下面的函数负责打印总的费用：

```
// 计算并打印销售给定数量的某种书籍所得的费用
double print_total(ostream &os,
                   const Quote &item, size_t n)
{
    // 根据传入 item 形参的对象类型调用 Quote::net_price
    // 或者 Bulk_quote::net_price
    double ret = item.net_price(n);
    os << "ISBN: " << item.isbn()     // 调用 Quote::isbn
       << " # sold: " << n << " total due: " << ret << endl;
    return ret;
}
```

该函数非常简单：它返回调用 net_price()的结果，并将该结果连同调用 isbn()的结果一起打印出来。

关于上面的函数有两个有意思的结论：因为函数 print_total 的 item 形参是基类 Quote 的一个引用，所以出于 15.2.3 节（第 534 页）将要解释的原因，我们既能使用基类 Quote 的对象调用该函数，也能使用派生类 Bulk_quote 的对象调用它；又因为 print_total 是使用引用类型调用 net_price 函数的，所以出于 15.2.1 节（第 528 页）将要解释的原因，实际传入 print_total 的对象类型将决定到底执行 net_price 的哪个版本：

```
// basic 的类型是 Quote; bulk 的类型是 Bulk_quote
print_total(cout, basic, 20);          // 调用 Quote 的 net_price
print_total(cout, bulk, 20);           // 调用 Bulk_quote 的 net_price
```

第一条调用句将 Quote 对象传入 print_total，因此当 print_total 调用 net_price 时，执行的是 Quote 的版本；在第二条调用语句中，实参的类型是 Bulk_quote，因此执行的是 Bulk_quote 的版本（计算打折信息）。因为在上述过程中函数的运行版本由实参决定，即在运行时选择函数的版本，所以动态绑定有时又被称为**运行时绑定**（run-time binding）。 594

在 C++语言中，当我们使用基类的引用（或指针）调用一个虚函数时将发生动态绑定。

15.2 定义基类和派生类

定义基类和派生类的方式在很多方面都与我们已知的定义其他类的方式类似，但是也有一些不同之处。本节将介绍在定义有继承关系的类时可能用到的基本特性。

15.2.1 定义基类

我们首先完成 Quote 类的定义：

```
class Quote {
public:
    Quote() = default;          // 关于=default 请参见 7.1.4 节（第 237 页）
    Quote(const std::string &book, double sales_price):
                        bookNo(book), price(sales_price) { }
    std::string isbn() const { return bookNo; }
    // 返回给定数量的书籍的销售总额
    // 派生类负责改写并使用不同的折扣计算算法
    virtual double net_price(std::size_t n) const
                { return n * price; }
    virtual ~Quote() = default;  // 对析构函数进行动态绑定
private:
    std::string bookNo;          // 书籍的 ISBN 编号
protected:
    double price = 0.0;          // 代表普通状态下不打折的价格
};
```

对于上面这个类来说，新增的部分是在 net_price 函数和析构函数之前增加的 virtual 关键字以及最后的 protected 访问说明符。我们将在 15.7.1 节（第 552 页）详细介绍虚析构函数的知识，现在只需记住作为继承关系中根节点的类通常都会定义一个虚析构函数。

基类通常都应该定义一个虚析构函数，即使该函数不执行任何实际操作也是如此。

成员函数与继承

派生类可以继承其基类的成员，然而当遇到如 net_price 这样与类型相关的操作时，派生类必须对其重新定义。换句话说，派生类需要对这些操作提供自己的新定义以**覆盖**（override）从基类继承而来的旧定义。

595

在 C++语言中，基类必须将它的两种成员函数区分开来：一种是基类希望其派生类进行覆盖的函数；另一种是基类希望派生类直接继承而不要改变的函数。对于前者，基类通常将其定义为**虚函数**（virtual）。当我们使用指针或引用调用虚函数时，该调用将被动态绑定。根据引用或指针所绑定的对象类型不同，该调用可能执行基类的版本，也可能执行某个派生类的版本。

基类通过在其成员函数的声明语句之前加上关键字 virtual 使得该函数执行动态绑定。任何构造函数之外的非静态函数（参见 7.6 节，第 268 页）都可以是虚函数。关键字 virtual 只能出现在类内部的声明语句之前而不能用于类外部的函数定义。如果基类把一个函数声明成虚函数，则该函数在派生类中隐式地也是虚函数。我们将在 15.3 节（第 536 页）介绍更多关于虚函数的知识。

成员函数如果没被声明为虚函数，则其解析过程发生在编译时而非运行时。对于 isbn 成员来说这正是我们希望看到的结果。isbn 函数的执行与派生类的细节无关，不管作用于 Quote 对象还是 Bulk_quote 对象，isbn 函数的行为都一样。在我们的继承层次关系中只有一个 isbn 函数，因此也就不存在调用 isbn()时到底执行哪个版本的疑问。

访问控制与继承

派生类可以继承定义在基类中的成员，但是派生类的成员函数不一定有权访问从基类继承而来的成员。和其他使用基类的代码一样，派生类能访问公有成员，而不能访问私有成员。不过在某些时候基类中还有这样一种成员，基类希望它的派生类有权访问该成员，同时禁止其他用户访问。我们用**受保护的**（protected）访问运算符说明这样的成员。

我们的 Quote 类希望它的派生类定义各自的 net_price 函数，因此派生类需要访问 Quote 的 price 成员。此时我们将 price 定义成受保护的。与之相反，派生类访问 bookNo 成员的方式与其他用户是一样的，都是通过调用 isbn 函数，因此 bookNo 被定义成私有的，即使是 Quote 派生出来的类也不能直接访问它。我们将在 15.5 节（第 542 页）介绍更多关于受保护成员的知识。

15.2.1 节练习

练习 15.1：什么是虚成员？

练习 15.2：protected 访问说明符与 private 有何区别？

练习 15.3：定义你自己的 Quote 类和 print_total 函数。

15.2.2 定义派生类

596

派生类必须通过使用**类派生列表**（class derivation list）明确指出它是从哪个（哪些）基类继承而来的。类派生列表的形式是：首先是一个冒号，后面紧跟以逗号分隔的基类列表，其中每个基类前面可以有以下三种访问说明符中的一个：public、protected 或者 private。

派生类必须将其继承而来的成员函数中需要覆盖的那些重新声明，因此，我们的 Bulk_quote 类必须包含一个 net_price 成员：

```
class Bulk_quote : public Quote {          // Bulk_quote 继承自 Quote
public:
    Bulk_quote() = default;
    Bulk_quote(const std::string&, double, std::size_t, double);
    // 覆盖基类的函数版本以实现基于大量购买的折扣政策
    double net_price(std::size_t) const override;
private:
    std::size_t min_qty = 0;               // 适用折扣政策的最低购买量
    double discount = 0.0;                 // 以小数表示的折扣额
};
```

我们的 Bulk_quote 类从它的基类 Quote 那里继承了 isbn 函数和 bookNo、price 等数据成员。此外，它还定义了 net_price 的新版本，同时拥有两个新增加的数据成员 min_qty 和 discount。这两个成员分别用于说明享受折扣所需购买的最低数量以及一旦该数量达到之后具体的折扣信息。

我们将在 15.5 节（第 543 页）详细介绍派生列表中用到的访问说明符。现在，我们只需知道访问说明符的作用是控制派生类从基类继承而来的成员是否对派生类的用户可见。

如果一个派生是公有的，则基类的公有成员也是派生类接口的组成部分。此外，我们能将公有派生类型的对象绑定到基类的引用或指针上。因为我们在派生列表中使用了

public，所以 Bulk_quote 的接口隐式地包含 isbn 函数，同时在任何需要 Quote 的引用或指针的地方我们都能使用 Bulk_quote 的对象。

大多数类都只继承自一个类，这种形式的继承被称作“单继承”，它构成了本章的主题。关于派生列表中含有多于一个基类的情况将在 18.3 节（第 710 页）中介绍。

派生类中的虚函数

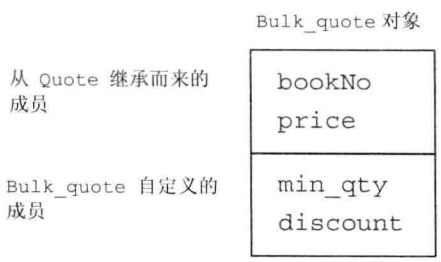
派生类经常（但不总是）覆盖它继承的虚函数。如果派生类没有覆盖其基类中的某个虚函数，则该虚函数的行为类似于其他的普通成员，派生类会直接继承其在基类中的版本。

C++ 11 派生类可以在它覆盖的函数前使用 virtual 关键字，但不是非得这么做。我们将在 15.3 节（第 538 页）介绍其原因，C++11 新标准允许派生类显式地注明它使用某个成员函数覆盖了它继承的虚函数。具体做法是在形参列表后面、或者在 const 成员函数（参见 7.1.2 节，第 231 页）的 const 关键字后面、或者在引用成员函数（参见 13.6.3 节，第 483 页）的引用限定符后面添加一个关键字 override。

597 ### 派生类对象及派生类向基类的类型转换

一个派生类对象包含多个组成部分：一个含有派生类自己定义的（非静态）成员的子对象，以及一个与该派生类继承的基类对应的子对象，如果有多个基类，那么这样的子对象也有多个。因此，一个 Bulk_quote 对象将包含四个数据元素：它从 Quote 继承而来的 bookNo 和 price 数据成员，以及 Bulk_quote 自己定义的 min_qty 和 discount 成员。

C++标准并没有明确规定派生类的对象在内存中如何分布，但是我们可以认为 Bulk_quote 的对象包含如图 15.1 所示的两部分。



在一个对象中，继承自基类的部分和派生类自定义的部分不一定是连续存储的。图 15.1 只是表示类工作机理的概念模型，而非物理模型。

图 15.1：Bulk_quote 对象的概念结构

因为在派生类对象中含有与其基类对应的组成部分，所以我们能把派生类的对象当成基类对象来使用，而且我们也能将基类的指针或引用绑定到派生类对象中的基类部分上。

```
Quote item;                 // 基类对象
Bulk_quote bulk;            // 派生类对象
Quote *p = &item;           // p 指向 Quote 对象
p = &bulk;                  // p 指向 bulk 的 Quote 部分
Quote &r = bulk;            // r 绑定到 bulk 的 Quote 部分
```

这种转换通常称为**派生类到基类的**（derived-to-base）类型转换。和其他类型转换一样，编译器会隐式地执行派生类到基类的转换（参见 4.11 节，第 141 页）。

这种隐式特性意味着我们可以把派生类对象或者派生类对象的引用用在需要基类引

用的地方；同样的，我们也可以把派生类对象的指针用在需要基类指针的地方。

在派生类对象中含有与其基类对应的组成部分，这一事实是继承的关键所在。

派生类构造函数

598

尽管在派生类对象中含有从基类继承而来的成员，但是派生类并不能直接初始化这些成员。和其他创建了基类对象的代码一样，派生类也必须使用基类的构造函数来初始化它的基类部分。

每个类控制它自己的成员初始化过程。

派生类对象的基类部分与派生类对象自己的数据成员都是在构造函数的初始化阶段（参见 7.5.1 节，第 258 页）执行初始化操作的。类似于我们初始化成员的过程，派生类构造函数同样是通过构造函数初始化列表来将实参传递给基类构造函数的。例如，接受四个参数的 Bulk_quote 构造函数如下所示：

```
Bulk_quote(const std::string& book, double p,
           std::size_t qty, double disc) :
           Quote(book, p), min_qty(qty), discount(disc) { }
    // 与之前一致
};
```

该函数将它的前两个参数（分别表示 ISBN 和价格）传递给 Quote 的构造函数，由 Quote 的构造函数负责初始化 Bulk_quote 的基类部分（即 bookNo 成员和 price 成员）。当（空的）Quote 构造函数体结束后，我们构建的对象的基类部分也就完成初始化了。接下来初始化由派生类直接定义的 min_qty 成员和 discount 成员。最后运行 Bulk_quote 构造函数的（空的）函数体。

除非我们特别指出，否则派生类对象的基类部分会像数据成员一样执行默认初始化。如果想使用其他的基类构造函数，我们需要以类名加圆括号内的实参列表的形式为构造函数提供初始值。这些实参将帮助编译器决定到底应该选用哪个构造函数来初始化派生类对象的基类部分。

首先初始化基类的部分，然后按照声明的顺序依次初始化派生类的成员。

派生类使用基类的成员

派生类可以访问基类的公有成员和受保护成员：

```
// 如果达到了购买书籍的某个最低限量值，就可以享受折扣价格了
double Bulk_quote::net_price(size_t cnt) const
{
    if (cnt >= min_qty)
        return cnt * (1 - discount) * price;
    else
        return cnt * price;
}
```

该函数产生一个打折后的价格：如果给定的数量超过了 min_qty，则将 discount（一

599

个小于 1 大于 0 的数）作用于 price。

我们将在 15.6 节（第 547 页）进一步讨论作用域，目前只需要了解派生类的作用域嵌套在基类的作用域之内。因此，对于派生类的一个成员来说，它使用派生类成员（例如 min_qty 和 discount）的方式与使用基类成员（例如 price）的方式没什么不同。

关键概念：遵循基类的接口

必须明确一点：每个类负责定义各自的接口。要想与类的对象交互必须使用该类的接口，即使这个对象是派生类的基类部分也是如此。

因此，派生类对象不能直接初始化基类的成员。尽管从语法上来说我们可以在派生类构造函数体内给它的公有或受保护的基类成员赋值，但是最好不要这么做。和使用基类的其他场合一样，派生类应该遵循基类的接口，并且通过调用基类的构造函数来初始化那些从基类中继承而来的成员。

继承与静态成员

如果基类定义了一个静态成员（参见 7.6 节，第 268 页），则在整个继承体系中只存在该成员的唯一定义。不论从基类中派生出来多少个派生类，对于每个静态成员来说都只存在唯一的实例。

```
class Base {
public:
    static void statmem();
};
class Derived : public Base {
    void f(const Derived&);
};
```

静态成员遵循通用的访问控制规则，如果基类中的成员是 private 的，则派生类无权访问它。假设某静态成员是可访问的，则我们既能通过基类使用它也能通过派生类使用它：

```
void Derived::f(const Derived &derived_obj)
{
    Base::statmem();                // 正确：Base 定义了 statmem
    Derived::statmem();             // 正确：Derived 继承了 statmem
    // 正确：派生类的对象能访问基类的静态成员
    derived_obj.statmem();          // 通过 Derived 对象访问
    statmem();                      // 通过 this 对象访问
}
```

600 派生类的声明

派生类的声明与其他类差别不大（参见 7.3.3 节，第 250 页），声明中包含类名但是不包含它的派生列表：

```
class Bulk_quote : public Quote;  // 错误：派生列表不能出现在这里
class Bulk_quote;                 // 正确：声明派生类的正确方式
```

一条声明语句的目的是令程序知晓某个名字的存在以及该名字表示一个什么样的实体，如一个类、一个函数或一个变量等。派生列表以及与定义有关的其他细节必须与类的主体一起出现。

被用作基类的类

如果我们想将某个类用作基类，则该类必须已经定义而非仅仅声明：

```
class Quote;                                    // 声明但未定义
// 错误：Quote 必须被定义
class Bulk_quote : public Quote { ... };
```

这一规定的原因显而易见：派生类中包含并且可以使用它从基类继承而来的成员，为了使用这些成员，派生类当然要知道它们是什么。因此该规定还有一层隐含的意思，即一个类不能派生它本身。

一个类是基类，同时它也可以是一个派生类：

```
class Base { /* ...*/ };
class D1: public Base { /* ...*/ };
class D2: public D1 { /* ...*/ };
```

在这个继承关系中，Base 是 D1 的**直接基类**（direct base），同时是 D2 的**间接基类**（indirect base）。直接基类出现在派生列表中，而间接基类由派生类通过其直接基类继承而来。

每个类都会继承直接基类的所有成员。对于一个最终的派生类来说，它会继承其直接基类的成员；该直接基类的成员又含有其基类的成员；依此类推直至继承链的顶端。因此，最终的派生类将包含它的直接基类的子对象以及每个间接基类的子对象。

防止继承的发生

有时我们会定义这样一种类，我们不希望其他类继承它，或者不想考虑它是否适合作为一个基类。为了实现这一目的，C++11 新标准提供了一种防止继承发生的方法，即在类名后跟一个关键字 final：

C++ 11

```
class NoDerived final { /* */ };          // NoDerived 不能作为基类
class Base { /* */ };
// Last 是 final 的；我们不能继承 Last
class Last final : Base { /* */ };        // Last 不能作为基类
class Bad : NoDerived { /* */ };          // 错误：NoDerived 是 final 的
class Bad2 : Last { /* */ };              // 错误：Last 是 final 的
```

15.2.2 节练习

601

练习 15.4：下面哪条声明语句是不正确的？请解释原因。

```
class Base { ... };
(a) class Derived : public Derived { ... };
(b) class Derived : private Base { ... };
(c) class Derived : public Base;
```

练习 15.5：定义你自己的 Bulk_quote 类。

练习 15.6：将 Quote 和 Bulk_quote 的对象传给 15.2.1 节（第 529 页）练习中的 print_total 函数，检查该函数是否正确。

练习 15.7：定义一个类使其实现一种数量受限的折扣策略，具体策略是：当购买书籍的数量不超过一个给定的限量时享受折扣，如果购买量一旦超过了限量，则超出的部分将以原价销售。

15.2.3 类型转换与继承

理解基类和派生类之间的类型转换是理解 C++语言面向对象编程的关键所在。

通常情况下，如果我们想把引用或指针绑定到一个对象上，则引用或指针的类型应与对象的类型一致（参见 2.3.1 节，第 46 页和 2.3.2 节，第 47 页），或者对象的类型含有一个可接受的 const 类型转换规则（参见 4.11.2 节，第 144 页）。存在继承关系的类是一个重要的例外：我们可以将基类的指针或引用绑定到派生类对象上。例如，我们可以用 Quote& 指向一个 Bulk_quote 对象，也可以把一个 Bulk_quote 对象的地址赋给一个 Quote*。

可以将基类的指针或引用绑定到派生类对象上有一层极为重要的含义：当使用基类的引用（或指针）时，实际上我们并不清楚该引用（或指针）所绑定对象的真实类型。该对象可能是基类的对象，也可能是派生类的对象。

和内置指针一样，智能指针类（参见 12.1 节，第 400 页）也支持派生类向基类的类型转换，这意味着我们可以将一个派生类对象的指针存储在一个基类的智能指针内。

静态类型与动态类型

当我们使用存在继承关系的类型时，必须将一个变量或其他表达式的**静态类型**（static type）与该表达式表示对象的**动态类型**（dynamic type）区分开来。表达式的静态类型在编译时总是已知的，它是变量声明时的类型或表达式生成的类型；动态类型则是变量或表达式表示的内存中的对象的类型。动态类型直到运行时才可知。

602

例如，当 print_total 调用 net_price 时（参见 15.1 节，第 527 页）：

```
double ret = item.net_price(n);
```

我们知道 item 的静态类型是 Quote&，它的动态类型则依赖于 item 绑定的实参，动态类型直到在运行时调用该函数时才会知道。如果我们传递一个 Bulk_quote 对象给 print_total，则 item 的静态类型将与它的动态类型不一致。如前所述，item 的静态类型是 Quote&，而在此例中它的动态类型则是 Bulk_quote。

如果表达式既不是引用也不是指针，则它的动态类型永远与静态类型一致。例如，Quote 类型的变量永远是一个 Quote 对象，我们无论如何都不能改变该变量对应的对象的类型。

基类的指针或引用的静态类型可能与其动态类型不一致，读者一定要理解其中的原因。

不存在从基类向派生类的隐式类型转换……

之所以存在派生类向基类的类型转换是因为每个派生类对象都包含一个基类部分，而基类的引用或指针可以绑定到该基类部分上。一个基类的对象既可以以独立的形式存在，也可以作为派生类对象的一部分存在。如果基类对象不是派生类对象的一部分，则它只含有基类定义的成员，而不含有派生类定义的成员。

因为一个基类的对象可能是派生类对象的一部分，也可能不是，所以不存在从基类向派生类的自动类型转换：

```
Quote base;
Bulk_quote* bulkP = &base;          // 错误：不能将基类转换成派生类
Bulk_quote& bulkRef = base;         // 错误：不能将基类转换成派生类
```

如果上述赋值是合法的，则我们有可能会使用 bulkP 或 bulkRef 访问 base 中本不存在的成员。

除此之外还有一种情况显得有点特别，即使一个基类指针或引用绑定在一个派生类对象上，我们也不能执行从基类向派生类的转换：

```
Bulk_quote bulk;
Quote *itemP = &bulk;                // 正确：动态类型是 Bulk_quote
Bulk_quote *bulkP = itemP;           // 错误：不能将基类转换成派生类
```

编译器在编译时无法确定某个特定的转换在运行时是否安全，这是因为编译器只能通过检查指针或引用的静态类型来推断该转换是否合法。如果在基类中含有一个或多个虚函数，我们可以使用 dynamic_cast（参见 19.2.1 节，第 730 页）请求一个类型转换，该转换的安全检查将在运行时执行。同样，如果我们已知某个基类向派生类的转换是安全的，则我们可以使用 static_cast（参见 4.11.3 节，第 144 页）来强制覆盖掉编译器的检查工作。

……在对象之间不存在类型转换

派生类向基类的自动类型转换只对指针或引用类型有效，在派生类类型和基类类型之间不存在这样的转换。很多时候，我们确实希望将派生类对象转换成它的基类类型，但是这种转换的实际发生过程往往与我们期望的有所差别。 603

请注意，当我们初始化或赋值一个类类型的对象时，实际上是在调用某个函数。当执行初始化时，我们调用构造函数（参见 13.1.1 节，第 440 页和 13.6.2 节，第 473 页）；而当执行赋值操作时，我们调用赋值运算符（参见 13.1.2 节，第 443 页和 13.6.2 节，第 474 页）。这些成员通常都包含一个参数，该参数的类型是类类型的 const 版本的引用。

因为这些成员接受引用作为参数，所以派生类向基类的转换允许我们给基类的拷贝/移动操作传递一个派生类的对象。这些操作不是虚函数。当我们给基类的构造函数传递一个派生类对象时，实际运行的构造函数是基类中定义的那个，显然该构造函数只能处理基类自己的成员。类似的，如果我们将一个派生类对象赋值给一个基类对象，则实际运行的赋值运算符也是基类中定义的那个，该运算符同样只能处理基类自己的成员。

例如，我们的书店类使用了合成版本的拷贝和赋值操作（参见 13.1.1 节，第 440 页和 13.1.2 节，第 444 页）。关于拷贝控制与继承的知识将在 15.7.2 节（第 552 页）做更详细的介绍，现在我们只需要知道合成版本会像其他类一样逐成员地执行拷贝或赋值操作：

```
Bulk_quote bulk;              // 派生类对象
Quote item(bulk);             // 使用 Quote::Quote(const Quote&) 构造函数
item = bulk;                  // 调用 Quote::operator=(const Quote&)
```

当构造 item 时，运行 Quote 的拷贝构造函数。该函数只能处理 bookNo 和 price 两个成员，它负责拷贝 bulk 中 Quote 部分的成员，同时忽略掉 bulk 中 Bulk_quote 部分的成员。类似的，对于将 bulk 赋值给 item 的操作来说，只有 bulk 中 Quote 部分的成员被赋值给 item。

因为在上述过程中会忽略 Bulk_quote 部分，所以我们可以说 bulk 的 Bulk_quote 部分被**切掉**（sliced down）了。

当我们用一个派生类对象为一个基类对象初始化或赋值时，只有该派生类对象中的基类部分会被拷贝、移动或赋值，它的派生类部分将被忽略掉。

15.2.3 节练习

练习 15.8：给出静态类型和动态类型的定义。

练习 15.9：在什么情况下表达式的静态类型可能与动态类型不同？请给出三个静态类型与动态类型不同的例子。

练习 15.10：回忆我们在 8.1 节（第 279 页）进行的讨论，解释第 284 页中将 `ifstream` 传递给 `Sales_data` 的 `read` 函数的程序是如何工作的。

关键概念：存在继承关系的类型之间的转换规则

要想理解在具有继承关系的类之间发生的类型转换，有三点非常重要：

- 从派生类向基类的类型转换只对指针或引用类型有效。
- 基类向派生类不存在隐式类型转换。
- 和任何其他成员一样，派生类向基类的类型转换也可能会由于访问受限而变得不可行。我们将在 15.5 节（第 544 页）详细介绍可访问性的问题。

尽管自动类型转换只对指针或引用类型有效，但是继承体系中的大多数类仍然（显式或隐式地）定义了拷贝控制成员（参见第 13 章）。因此，我们通常能够将一个派生类对象拷贝、移动或赋值给一个基类对象。不过需要注意的是，这种操作只处理派生类对象的基类部分。

15.3 虚函数

如前所述，在 C++语言中，当我们使用基类的引用或指针调用一个虚成员函数时会执行动态绑定（参见 15.1 节，第 527 页）。因为我们直到运行时才能知道到底调用了哪个版本的虚函数，所以所有虚函数都必须有定义。通常情况下，如果我们不使用某个函数，则无须为该函数提供定义（参见 6.1.2 节，第 186 页）。但是我们必须为每一个虚函数都提供定义，而不管它是否被用到了，这是因为连编译器也无法确定到底会使用哪个虚函数。

604

对虚函数的调用可能在运行时才被解析

当某个虚函数通过指针或引用调用时，编译器产生的代码直到运行时才能确定应该调用哪个版本的函数。被调用的函数是与绑定到指针或引用上的对象的动态类型相匹配的那一个。

举个例子，考虑 15.1 节（第 527 页）的 `print_total` 函数，该函数通过其名为 `item` 的参数来进一步调用 `net_price`，其中 `item` 的类型是 `Quote&`。因为 `item` 是引用而且 `net_price` 是虚函数，所以我们到底调用 `net_price` 的哪个版本完全依赖于运行时绑定到 `item` 的实参的实际（动态）类型：

```
Quote base("0-201-82470-1", 50);
print_total(cout, base, 10);          // 调用 Quote::net_price
Bulk_quote derived("0-201-82470-1", 50, 5, .19);
```

```
print_total(cout, derived, 10);        // 调用 Bulk_quote::net_price
```

在第一条调用语句中，item 绑定到 Quote 类型的对象上，因此当 print_total 调用 net_price 时，运行在 Quote 中定义的版本。在第二条调用语句中，item 绑定到 Bulk_quote 类型的对象上，因此 print_total 调用 Bulk_quote 定义的 net_price。 605

必须要搞清楚的一点是，动态绑定只有当我们通过指针或引用调用虚函数时才会发生。

```
base = derived;                    // 把 derived 的 Quote 部分拷贝给 base
base.net_price(20);                // 调用 Quote::net_price
```

当我们通过一个具有普通类型（非引用非指针）的表达式调用虚函数时，在编译时就会将调用的版本确定下来。例如，如果我们使用 base 调用 net_price，则应该运行 net_price 的哪个版本是显而易见的。我们可以改变 base 表示的对象的值（即内容），但是不会改变该对象的类型。因此，在编译时该调用就会被解析成 Quote 的 net_price。

关键概念：C++的多态性

OOP 的核心思想是多态性（polymorphism）。多态性这个词源自希腊语，其含义是“多种形式”。我们把具有继承关系的多个类型称为多态类型，因为我们能使用这些类型的“多种形式”而无须在意它们的差异。引用或指针的静态类型与动态类型不同这一事实正是 C++语言支持多态性的根本所在。

当我们使用基类的引用或指针调用基类中定义的一个函数时，我们并不知道该函数真正作用的对象是什么类型，因为它可能是一个基类的对象也可能是一个派生类的对象。如果该函数是虚函数，则直到运行时才会决定到底执行哪个版本，判断的依据是引用或指针所绑定的对象的真实类型。

另一方面，对非虚函数的调用在编译时进行绑定。类似的，通过对象进行的函数（虚函数或非虚函数）调用也在编译时绑定。对象的类型是确定不变的，我们无论如何都不可能令对象的动态类型与静态类型不一致。因此，通过对象进行的函数调用将在编译时绑定到该对象所属类中的函数版本上。

当且仅当对通过指针或引用调用虚函数时，才会在运行时解析该调用，也只有在这种情况下对象的动态类型才有可能与静态类型不同。

派生类中的虚函数

当我们在派生类中覆盖了某个虚函数时，可以再一次使用 virtual 关键字指出该函数的性质。然而这么做并非必须，因为一旦某个函数被声明成虚函数，则在所有派生类中它都是虚函数。

一个派生类的函数如果覆盖了某个继承而来的虚函数，则它的形参类型必须与被它覆盖的基类函数完全一致。

同样，派生类中虚函数的返回类型也必须与基类函数匹配。该规则存在一个例外，当类的虚函数返回类型是类本身的指针或引用时，上述规则无效。也就是说，如果 D 由 B 派生得到，则基类的虚函数可以返回 B*而派生类的对应函数可以返回 D*，只不过这样的返回类型要求从 D 到 B 的类型转换是可访问的。15.5 节（第 544 页）将介绍如何确定一个基类的可访问性，在 15.8.1 节（第 561 页）中我们将看到这种虚函数的一个实际例子。 606

基类中的虚函数在派生类中隐含地也是一个虚函数。当派生类覆盖了某个虚函数时，该函数在基类中的形参必须与派生类中的形参严格匹配。

final 和 override 说明符

如我们将要在 15.6 节（第 550 页）介绍的，派生类如果定义了一个函数与基类中虚函数的名字相同但是形参列表不同，这仍然是合法的行为。编译器将认为新定义的这个函数与基类中原有的函数是相互独立的。这时，派生类的函数并没有覆盖掉基类中的版本。就实际的编程习惯而言，这种声明往往意味着发生了错误，因为我们可能原本希望派生类能覆盖掉基类中的虚函数，但是一不小心把形参列表弄错了。

C++ 11

要想调试并发现这样的错误显然非常困难。在 C++11 新标准中我们可以使用 override 关键字来说明派生类中的虚函数。这么做的好处是在使得程序员的意图更加清晰的同时让编译器可以为我们发现一些错误，后者在编程实践中显得更加重要。如果我们使用 override 标记了某个函数，但该函数并没有覆盖已存在的虚函数，此时编译器将报错：

```
struct B {
    virtual void f1(int) const;
    virtual void f2();
    void f3();
};
struct D1 : B {
    void f1(int) const override;      // 正确：f1 与基类中的 f1 匹配
    void f2(int) override;            // 错误：B 没有形如 f2(int) 的函数
    void f3() override;               // 错误：f3 不是虚函数
    void f4() override;               // 错误：B 没有名为 f4 的函数
};
```

在 D1 中，f1 的 override 说明符是正确的，因为基类和派生类中的 f1 都是 const 成员，并且它们都接受一个 int 返回 void，所以 D1 中的 f1 正确地覆盖了它从 B 中继承而来的虚函数。

D1 中 f2 的声明与 B 中 f2 的声明不匹配，显然 B 中定义的 f2 不接受任何参数而 D1 的 f2 接受一个 int。因为这两个声明不匹配，所以 D1 的 f2 不能覆盖 B 的 f2，它是一个新函数，仅仅是名字恰好与原来的函数一样而已。因为我们使用 override 所表达的意思是我们希望能覆盖基类中的虚函数而实际上并未做到，所以编译器会报错。

因为只有虚函数才能被覆盖，所以编译器会拒绝 D1 的 f3。该函数不是 B 中的虚函数，因此它不能被覆盖。类似的，f4 的声明也会发生错误，因为 B 中根本就没有名为 f4 的函数。

607

我们还能把某个函数指定为 final，如果我们已经把函数定义成 final 了，则之后任何尝试覆盖该函数的操作都将引发错误：

```
struct D2 : B {
    // 从 B 继承 f2() 和 f3()，覆盖 f1(int)
    void f1(int) const final;     // 不允许后续的其他类覆盖 f1(int)
};
struct D3 : D2 {
    void f2();                    // 正确：覆盖从间接基类 B 继承而来的 f2
    void f1(int) const;           // 错误：D2 已经将 f2 声明成 final
};
```

final 和 override 说明符出现在形参列表（包括任何 const 或引用修饰符）以及尾置返回类型（参见 6.3.3 节，第 206 页）之后。

虚函数与默认实参

和其他函数一样，虚函数也可以拥有默认实参（参见 6.5.1 节，第 211 页）。如果某次函数调用使用了默认实参，则该实参值由本次调用的静态类型决定。

换句话说，如果我们通过基类的引用或指针调用函数，则使用基类中定义的默认实参，即使实际运行的是派生类中的函数版本也是如此。此时，传入派生类函数的将是基类函数定义的默认实参。如果派生类函数依赖不同的实参，则程序结果将与我们的预期不符。

如果虚函数使用默认实参，则基类和派生类中定义的默认实参最好一致。

回避虚函数的机制

在某些情况下，我们希望对虚函数的调用不要进行动态绑定，而是强迫其执行虚函数的某个特定版本。使用作用域运算符可以实现这一目的，例如下面的代码：

```
// 强行调用基类中定义的函数版本而不管 baseP 的动态类型到底是什么
double undiscounted = baseP->Quote::net_price(42);
```

该代码强行调用 Quote 的 net_price 函数，而不管 baseP 实际指向的对象类型到底是什么。该调用将在编译时完成解析。

通常情况下，只有成员函数（或友元）中的代码才需要使用作用域运算符来回避虚函数的机制。

什么时候我们需要回避虚函数的默认机制呢？通常是当一个派生类的虚函数调用它覆盖的基类的虚函数版本时。在此情况下，基类的版本通常完成继承层次中所有类型都要做的共同任务，而派生类中定义的版本需要执行一些与派生类本身密切相关的操作。

如果一个派生类虚函数需要调用它的基类版本，但是没有使用作用域运算符，则在运行时该调用将被解析为对派生类版本自身的调用，从而导致无限递归。

608

15.3 节练习

练习 15.11：为你的 Quote 类体系添加一个名为 debug 的虚函数，令其分别显示每个类的数据成员。

练习 15.12：有必要将一个成员函数同时声明成 override 和 final 吗？为什么？

练习 15.13：给定下面的类，解释每个 print 函数的机理：

```
class base {
public:
    string name() { return basename; }
    virtual void print(ostream &os) { os << basename; }
private:
```

```
    string basename;
};
class derived : public base {
public:
    void print(ostream &os) { print(os); os << " " << i; }
private:
    int i;
};
```

在上述代码中存在问题吗？如果有，你该如何修改它？

练习 15.14：给定上一题中的类以及下面这些对象，说明在运行时调用哪个函数：

```
base bobj;        base *bp1 = &bobj;        base &br1 = bobj;
derived dobj;     base *bp2 = &dobj;        base &br2 = dobj;
(a) bobj.print();     (b) dobj.print();     (c) bp1->name();
(d) bp2->name();      (e) br1.print();      (f) br2.print();
```

15.4 抽象基类

假设我们希望扩展书店程序并令其支持几种不同的折扣策略。除了购买量超过一定数量享受折扣外，我们也可能提供另外一种策略，即购买量不超过某个限额时可以享受折扣，但是一旦超过限额就要按原价支付。或者折扣策略还可能是购买量超过一定数量后购买的全部书籍都享受折扣，否则全都不打折。

上面的每个策略都要求一个购买量的值和一个折扣值。我们可以定义一个新的名为 Disc_quote 的类来支持不同的折扣策略，其中 Disc_quote 负责保存购买量的值和折扣值。其他的表示某种特定策略的类（如 Bulk_quote）将分别继承自 Disc_quote，每个派生类通过定义自己的 net_price 函数来实现各自的折扣策略。

609

在定义 Disc_quote 类之前，首先要确定它的 net_price 函数完成什么工作。显然我们的 Disc_quote 类与任何特定的折扣策略都无关，因此 Disc_quote 类中的 net_price 函数是没有实际含义的。

我们可以在 Disc_quote 类中不定义新的 net_price，此时，Disc_quote 将继承 Quote 中的 net_price 函数。

然而，这样的设计可能导致用户编写出一些无意义的代码。用户可能会创建一个 Disc_quote 对象并为其提供购买量和折扣值，如果将该对象传给一个像 print_total 这样的函数，则程序将调用 Quote 版本的 net_price。显然，最终计算出的销售价格并没有考虑我们在创建对象时提供的折扣值，因此上述操作毫无意义。

纯虚函数

认真思考上面描述的情形我们可以发现，关键问题并不仅仅是不知道应该如何定义 net_price，而是我们根本就不希望用户创建一个 Disc_quote 对象。Disc_quote 类表示的是一本打折书籍的通用概念，而非某种具体的折扣策略。

我们可以将 net_price 定义成**纯虚**（pure virtual）函数从而令程序实现我们的设计意图，这样做可以清晰明了地告诉用户当前这个 net_price 函数是没有实际意义的。和普通的虚函数不一样，一个纯虚函数无须定义。我们通过在函数体的位置（即在声明语句

的分号之前）书写=0 就可以将一个虚函数说明为纯虚函数。其中，=0 只能出现在类内部的虚函数声明语句处：

```
// 用于保存折扣值和购买量的类，派生类使用这些数据可以实现不同的价格策略
class Disc_quote : public Quote {
public:
    Disc_quote() = default;
    Disc_quote(const std::string& book, double price,
               std::size_t qty, double disc):
                  Quote(book, price),
                  quantity(qty), discount(disc) { }
    double net_price(std::size_t) const = 0;
protected:
    std::size_t quantity = 0;          // 折扣适用的购买量
    double discount = 0.0;             // 表示折扣的小数值
};
```

和我们之前定义的 Bulk_quote 类一样，Disc_quote 也分别定义了一个默认构造函数和一个接受四个参数的构造函数。尽管我们不能直接定义这个类的对象，但是 Disc_quote 的派生类构造函数将会使用 Disc_quote 的构造函数来构建各个派生类对象的 Disc_quote 部分。其中，接受四个参数的构造函数将前两个参数传递给 Quote 的构造函数，然后直接初始化自己的成员 discount 和 quantity。默认构造函数则对这些成员进行默认初始化。

值得注意的是，我们也可以为纯虚函数提供定义，不过函数体必须定义在类的外部。也就是说，我们不能在类的内部为一个=0 的函数提供函数体。 610

含有纯虚函数的类是抽象基类

含有（或者未经覆盖直接继承）纯虚函数的类是**抽象基类**（abstract base class）。抽象基类负责定义接口，而后续的其他类可以覆盖该接口。我们不能（直接）创建一个抽象基类的对象。因为 Disc_quote 将 net_price 定义成了纯虚函数，所以我们不能定义 Disc_quote 的对象。我们可以定义 Disc_quote 的派生类的对象，前提是这些类覆盖了 net_price 函数：

```
// Disc_quote 声明了纯虚函数，而 Bulk_quote 将覆盖该函数
Disc_quote discounted;          // 错误：不能定义 Disc_quote 的对象
Bulk_quote bulk;                // 正确：Bulk_quote 中没有纯虚函数
```

Disc_quote 的派生类必须给出自己的 net_price 定义，否则它们仍将是抽象基类。

我们不能创建抽象基类的对象。

派生类构造函数只初始化它的直接基类

接下来可以重新实现 Bulk_quote 了，这一次我们让它继承 Disc_quote 而非直接继承 Quote：

```
// 当同一书籍的销售量超过某个值时启用折扣
// 折扣的值是一个小于 1 的正的小数值，以此来降低正常销售价格
class Bulk_quote : public Disc_quote {
public:
    Bulk_quote() = default;
```

```
    Bulk_quote(const std::string& book, double price,
               std::size_t qty, double disc):
        Disc_quote(book, price, qty, disc) { }
    // 覆盖基类中的函数版本以实现一种新的折扣策略
    double net_price(std::size_t) const override;
};
```

这个版本的 Bulk_quote 的直接基类是 Disc_quote，间接基类是 Quote。每个 Bulk_quote 对象包含三个子对象：一个（空的）Bulk_quote 部分、一个 Disc_quote 子对象和一个 Quote 子对象。

如前所述，每个类各自控制其对象的初始化过程。因此，即使 Bulk_quote 没有自己的数据成员，它也仍然需要像原来一样提供一个接受四个参数的构造函数。该构造函数将它的实参传递给 Disc_quote 的构造函数，随后 Disc_quote 的构造函数继续调用 Quote 的构造函数。Quote 的构造函数首先初始化 bulk 的 bookNo 和 price 成员，当 Quote 的构造函数结束后，开始运行 Disc_quote 的构造函数并初始化 quantity 和 discount 成员，最后运行 Bulk_quote 的构造函数，该函数无须执行实际的初始化或其他工作。

611

关键概念：重构

在 Quote 的继承体系中增加 Disc_quote 类是重构（refactoring）的一个典型示例。重构负责重新设计类的体系以便将操作和/或数据从一个类移动到另一个类中。对于面向对象的应用程序来说，重构是一种很普遍的现象。

值得注意的是，即使我们改变了整个继承体系，那些使用了 Bulk_quote 或 Quote 的代码也无须进行任何改动。不过一旦类被重构（或以其他方式被改变），就意味着我们必须重新编译含有这些类的代码了。

15.4 节练习

练习 15.15：定义你自己的 Disc_quote 和 Bulk_quote。

练习 15.16：改写你在 15.2.2 节（第 533 页）练习中编写的数量受限的折扣策略，令其继承 Disc_quote。

练习 15.17：尝试定义一个 Disc_quote 的对象，看看编译器给出的错误信息是什么？

15.5 访问控制与继承

每个类分别控制自己的成员初始化过程（参见 15.2.2 节，第 531 页），与之类似，每个类还分别控制着其成员对于派生类来说是否**可访问**（accessible）。

受保护的成员

如前所述，一个类使用 protected 关键字来声明那些它希望与派生类分享但是不想被其他公共访问使用的成员。protected 说明符可以看做是 public 和 private 中和后的产物：

- 和私有成员类似，受保护的成员对于类的用户来说是不可访问的。

- 和公有成员类似，受保护的成员对于派生类的成员和友元来说是可访问的。

此外，protected 还有另外一条重要的性质。

- 派生类的成员或友元只能通过派生类对象来访问基类的受保护成员。派生类对于一个基类对象中的受保护成员没有任何访问特权。

为了理解最后一条规则，请考虑如下的例子：

612

```
class Base {
protected:
    int prot_mem;                       // protected 成员
};
class Sneaky : public Base {
    friend void clobber(Sneaky&);       // 能访问 Sneaky::prot_mem
    friend void clobber(Base&);         // 不能访问 Base::prot_mem
    int j;                              // j 默认是 private
};
// 正确：clobber 能访问 Sneaky 对象的 private 和 protected 成员
void clobber(Sneaky &s) { s.j = s.prot_mem = 0; }
// 错误：clobber 不能访问 Base 的 protected 成员
void clobber(Base &b) { b.prot_mem = 0; }
```

如果派生类（及其友元）能访问基类对象的受保护成员，则上面的第二个 clobber（接受一个 Base&）将是合法的。该函数不是 Base 的友元，但是它仍然能够改变一个 Base 对象的内容。如果按照这样的思路，则我们只要定义一个形如 Sneaky 的新类就能非常简单地规避掉 protected 提供的访问保护了。

要想阻止以上的用法，我们就要做出如下规定，即派生类的成员和友元只能访问派生类对象中的基类部分的受保护成员；对于普通的基类对象中的成员不具有特殊的访问权限。

公有、私有和受保护继承

某个类对其继承而来的成员的访问权限受到两个因素影响：一是在基类中该成员的访问说明符，二是在派生类的派生列表中的访问说明符。举个例子，考虑如下的继承关系：

```
class Base {
public:
    void pub_mem();                 // public 成员
protected:
    int prot_mem;                   // protected 成员
private:
    char priv_mem;                  // private 成员
};
struct Pub_Derv : public Base {
    // 正确：派生类能访问 protected 成员
    int f() { return prot_mem; }
    // 错误：private 成员对于派生类来说是不可访问的
    char g() { return priv_mem; }
};
struct Priv_Derv : private Base {
    // private 不影响派生类的访问权限
    int f1() const { return prot_mem; }
};
```

派生访问说明符对于派生类的成员（及友元）能否访问其直接基类的成员没什么影响。对基类成员的访问权限只与基类中的访问说明符有关。Pub_Derv 和 Priv_Derv 都能访问受保护的成员 prot_mem，同时它们都不能访问私有成员 priv_mem。

613

派生访问说明符的目的是控制派生类用户（包括派生类的派生类在内）对于基类成员的访问权限：

```
Pub_Derv d1;                    // 继承自 Base 的成员是 public 的
Priv_Derv d2;                   // 继承自 Base 的成员是 private 的
d1.pub_mem();                   // 正确：pub_mem 在派生类中是 public 的
d2.pub_mem();                   // 错误：pub_mem 在派生类中是 private 的
```

Pub_Derv 和 Priv_Derv 都继承了 pub_mem 函数。如果继承是公有的，则成员将遵循其原有的访问说明符，此时 d1 可以调用 pub_mem。在 Priv_Derv 中，Base 的成员是私有的，因此类的用户不能调用 pub_mem。

派生访问说明符还可以控制继承自派生类的新类的访问权限：

```
struct Derived_from_Public : public Pub_Derv {
    // 正确：Base::prot_mem 在 Pub_Derv 中仍然是 protected 的
    int use_base() { return prot_mem; }
};
struct Derived_from_Private : public Priv_Derv {
    // 错误：Base::prot_mem 在 Priv_Derv 中是 private 的
    int use_base() { return prot_mem; }
};
```

Pub_Derv 的派生类之所以能访问 Base 的 prot_mem 成员是因为该成员在 Pub_Derv 中仍然是受保护的。相反，Priv_Derv 的派生类无法执行类的访问，对于它们来说，Priv_Derv 继承自 Base 的所有成员都是私有的。

假设我们之前还定义了一个名为 Prot_Derv 的类，它采用受保护继承，则 Base 的所有公有成员在新定义的类中都是受保护的。Prot_Derv 的用户不能访问 pub_mem，但是 Prot_Derv 的成员和友元可以访问那些继承而来的成员。

派生类向基类转换的可访问性

派生类向基类的转换（参见 15.2.2 节，第 530 页）是否可访问由使用该转换的代码决定，同时派生类的派生访问说明符也会有影响。假定 D 继承自 B：

- 只有当 D 公有地继承 B 时，用户代码才能使用派生类向基类的转换；如果 D 继承 B 的方式是受保护的或者私有的，则用户代码不能使用该转换。
- 不论 D 以什么方式继承 B，D 的成员函数和友元都能使用派生类向基类的转换；派生类向其直接基类的类型转换对于派生类的成员和友元来说永远是可访问的。
- 如果 D 继承 B 的方式是公有的或者受保护的，则 D 的派生类的成员和友元可以使用 D 向 B 的类型转换；反之，如果 D 继承 B 的方式是私有的，则不能使用。

614

对于代码中的某个给定节点来说，如果基类的公有成员是可访问的，则派生类向基类的类型转换也是可访问的；反之则不行。

关键概念：类的设计与受保护的成员

不考虑继承的话，我们可以认为一个类有两种不同的用户：普通用户和类的实现者。

其中，普通用户编写的代码使用类的对象，这部分代码只能访问类的公有（接口）成员；实现者则负责编写类的成员和友元的代码，成员和友元既能访问类的公有部分，也能访问类的私有（实现）部分。

如果进一步考虑继承的话就会出现第三种用户，即派生类。基类把它希望派生类能够使用的部分声明成受保护的。普通用户不能访问受保护的成员，而派生类及其友元仍旧不能访问私有成员。

和其他类一样，基类应该将其接口成员声明为公有的；同时将属于其实现的部分分成两组：一组可供派生类访问，另一组只能由基类及基类的友元访问。对于前者应该声明为受保护的，这样派生类就能在实现自己的功能时使用基类的这些操作和数据；对于后者应该声明为私有的。

友元与继承

就像友元关系不能传递一样（参见 7.3.4 节，第 250 页），友元关系同样也不能继承。基类的友元在访问派生类成员时不具有特殊性，类似的，派生类的友元也不能随意访问基类的成员：

```
class Base {
    // 添加 friend 声明，其他成员与之前的版本一致
    friend class Pal;              // Pal 在访问 Base 的派生类时不具有特殊性
};
class Pal {
public:
    int f(Base b) { return b.prot_mem; } // 正确：Pal 是 Base 的友元
    int f2(Sneaky s) { return s.j; }     // 错误：Pal 不是 Sneaky 的友元
    // 对基类的访问权限由基类本身控制，即使对于派生类的基类部分也是如此
    int f3(Sneaky s) { return s.prot_mem; }  // 正确：Pal 是 Base 的友元
};
```

如前所述，每个类负责控制自己的成员的访问权限，因此尽管看起来有点儿奇怪，但 f3 确实是正确的。Pal 是 Base 的友元，所以 Pal 能够访问 Base 对象的成员，这种可访问性包括了 Base 对象内嵌在其派生类对象中的情况。 615

当一个类将另一个类声明为友元时，这种友元关系只对做出声明的类有效。对于原来那个类来说，其友元的基类或者派生类不具有特殊的访问能力：

```
// D2 对 Base 的 protected 和 private 成员不具有特殊的访问能力
class D2 : public Pal {
public:
    int mem(Base b)
        { return b.prot_mem; }          // 错误：友元关系不能继承
};
```

不能继承友元关系；每个类负责控制各自成员的访问权限。

改变个别成员的可访问性

有时我们需要改变派生类继承的某个名字的访问级别，通过使用 using 声明（参见 3.1 节，第 74 页）可以达到这一目的：

```
class Base {
public:
```

```
    std::size_t size() const { return n; }
protected:
    std::size_t n;
};
class Derived : private Base {              // 注意: private 继承
public:
    // 保持对象尺寸相关的成员的访问级别
    using Base::size;
protected:
    using Base::n;
};
```

因为 Derived 使用了私有继承，所以继承而来的成员 size 和 n（在默认情况下）是 Derived 的私有成员。然而，我们使用 using 声明语句改变了这些成员的可访问性。改变之后，Derived 的用户将可以使用 size 成员，而 Derived 的派生类将能使用 n。

通过在类的内部使用 using 声明语句，我们可以将该类的直接或间接基类中的任何可访问成员（例如，非私有成员）标记出来。using 声明语句中名字的访问权限由该 using 声明语句之前的访问说明符来决定。也就是说，如果一条 using 声明语句出现在类的 private 部分，则该名字只能被类的成员和友元访问；如果 using 声明语句位于 public 部分，则类的所有用户都能访问它；如果 using 声明语句位于 protected 部分，则该名字对于成员、友元和派生类是可访问的。

派生类只能为那些它可以访问的名字提供 using 声明。

616 **默认的继承保护级别**

在 7.2 节（第 240 页）中我们曾经介绍过使用 struct 和 class 关键字定义的类具有不同的默认访问说明符。类似的，默认派生运算符也由定义派生类所用的关键字来决定。默认情况下，使用 class 关键字定义的派生类是私有继承的；而使用 struct 关键字定义的派生类是公有继承的：

```
class Base { /* ...*/ };
struct D1 : Base { /* ...*/ };                    // 默认 public 继承
class D2 : Base { /* ...*/ };                     // 默认 private 继承
```

人们常常有一种错觉，认为在使用 struct 关键字和 class 关键字定义的类之间还有更深层次的差别。事实上，唯一的差别就是默认成员访问说明符及默认派生访问说明符；除此之外，再无其他不同之处。

一个私有派生的类最好显式地将 private 声明出来，而不要仅仅依赖于默认的设置。显式声明的好处是可以令私有继承关系清晰明了，不至于产生误会。

15.5 节练习

练习 15.18：假设给定了第 543 页和第 544 页的类，同时已知每个对象的类型如注释所示，判断下面的哪些赋值语句是合法的。解释那些不合法的语句为什么不被允许：

```
Base *p = &d1;                  // d1 的类型是 Pub_Derv
p = &d2;                        // d2 的类型是 Priv_Derv
```

```
p = &d3;                    // d3 的类型是 Prot_Derv
p = &dd1;                   // dd1 的类型是 Derived_from_Public
p = &dd2;                   // dd2 的类型是 Derived_from_Private
p = &dd3;                   // dd3 的类型是 Derived_from_Protected
```

练习 15.19：假设 543 页和 544 页的每个类都有如下形式的成员函数：

```
void memfcn(Base &b) { b = *this; }
```

对于每个类，分别判断上面的函数是否合法。

练习 15.20：编写代码检验你对前面两题的回答是否正确。

练习 15.21：从下面这些一般性抽象概念中任选一个（或者选一个你自己的），将其对应的一组类型组织成一个继承体系：

(a) 图形文件格式（如 gif、tiff、jpeg、bmp）

(b) 图形基元（如方格、圆、球、圆锥）

(c) C++语言中的类型（如类、函数、成员函数）

练习 15.22：对于你在上一题中选择的类，为其添加合适的虚函数及公有成员和受保护的成员。

15.6 继承中的类作用域

每个类定义自己的作用域（参见 7.4 节，第 253 页），在这个作用域内我们定义类的成员。当存在继承关系时，派生类的作用域嵌套（参见 2.2.4 节，第 43 页）在其基类的作用域之内。如果一个名字在派生类的作用域内无法正确解析，则编译器将继续在外层的基类作用域中寻找该名字的定义。 617

派生类的作用域位于基类作用域之内这一事实可能有点儿出人意料，毕竟在我们的程序文本中派生类和基类的定义是相互分离开来的。不过也恰恰因为类作用域有这种继承嵌套的关系，所以派生类才能像使用自己的成员一样使用基类的成员。例如，当我们编写下面的代码时：

```
Bulk_quote bulk;
cout << bulk.isbn();
```

名字 isbn 的解析将按照下述过程所示：

- 因为我们是通过 Bulk_quote 的对象调用 isbn 的，所以首先在 Bulk_quote 中查找，这一步没有找到名字 isbn。
- 因为 Bulk_quote 是 Disc_quote 的派生类，所以接下来在 Disc_quote 中查找，仍然找不到。
- 因为 Disc_quote 是 Quote 的派生类，所以接着查找 Quote；此时找到了名字 isbn，所以我们使用的 isbn 最终被解析为 Quote 中的 isbn。

在编译时进行名字查找

一个对象、引用或指针的静态类型（参见 15.2.3 节，第 532 页）决定了该对象的哪些成员是可见的。即使静态类型与动态类型可能不一致（当使用基类的引用或指针时会发生

这种情况），但是我们能使用哪些成员仍然是由静态类型决定的。举个例子，我们可以给 Disc_quote 添加一个新成员，该成员返回一个存有最小（或最大）数量及折扣价格的 pair（参见 11.2.3 节，第 379 页）：

```
class Disc_quote : public Quote {
public:
    std::pair<size_t, double> discount_policy() const
        { return {quantity, discount}; }
    // 其他成员与之前的版本一致
};
```

我们只能通过 Disc_quote 及其派生类的对象、引用或指针使用 discount_policy：

```
Bulk_quote bulk;
Bulk_quote *bulkP = &bulk;          // 静态类型与动态类型一致
Quote *itemP = &bulk;               // 静态类型与动态类型不一致
bulkP->discount_policy();           // 正确：bulkP 的类型是 Bulk_quote*
itemP->discount_policy();           // 错误：itemP 的类型是 Quote*
```

618 尽管在 bulk 中确实含有一个名为 discount_policy 的成员，但是该成员对于 itemP 却是不可见的。itemP 的类型是 Quote 的指针，意味着对 discount_policy 的搜索将从 Quote 开始。显然 Quote 不包含名为 discount_policy 的成员，所以我们无法通过 Quote 的对象、引用或指针调用 discount_policy。

名字冲突与继承

和其他作用域一样，派生类也能重用定义在其直接基类或间接基类中的名字，此时定义在内层作用域（即派生类）的名字将隐藏定义在外层作用域（即基类）的名字（参见 2.2.4 节，第 43 页）：

```
struct Base {
    Base(): mem(0) { }
protected:
    int mem;
};
struct Derived : Base {
    Derived(int i): mem(i) { }              // 用 i 初始化 Derived::mem
                                            // Base::mem 进行默认初始化
    int get_mem() { return mem; }           // 返回 Derived::mem
protected:
    int mem;                                // 隐藏基类中的 mem
};
```

get_mem 中 mem 引用的解析结果是定义在 Derived 中的名字，下面的代码

```
Derived d(42);
cout << d.get_mem() << endl;          // 打印 42
```

的输出结果将是 42。

派生类的成员将隐藏同名的基类成员。

通过作用域运算符来使用隐藏的成员

我们可以通过作用域运算符来使用一个被隐藏的基类成员：

```
struct Derived : Base {
    int get_base_mem() { return Base::mem; }
    // ...
};
```

作用域运算符将覆盖掉原有的查找规则，并指示编译器从 Base 类的作用域开始查找 mem。如果使用最新的 Derived 版本运行上面的代码，则 d.get_mem() 的输出结果将是 0。

Best Practices 除了覆盖继承而来的虚函数之外，派生类最好不要重用其他定义在基类中的名字。

关键概念：名字查找与继承 619

理解函数调用的解析过程对于理解 C++的继承至关重要，假定我们调用 p->mem()（或者 obj.mem()），则依次执行以下 4 个步骤：

- 首先确定 p（或 obj）的静态类型。因为我们调用的是一个成员，所以该类型必然是类类型。
- 在 p（或 obj）的静态类型对应的类中查找 mem。如果找不到，则依次在直接基类中不断查找直至到达继承链的顶端。如果找遍了该类及其基类仍然找不到，则编译器将报错。
- 一旦找到了 mem，就进行常规的类型检查（参见 6.1 节，第 183 页）以确认对于当前找到的 mem，本次调用是否合法。
- 假设调用合法，则编译器将根据调用的是否是虚函数而产生不同的代码：
  - 如果 mem 是虚函数且我们是通过引用或指针进行的调用，则编译器产生的代码将在运行时确定到底运行该虚函数的哪个版本，依据是对象的动态类型。
  - 反之，如果 mem 不是虚函数或者我们是通过对象（而非引用或指针）进行的调用，则编译器将产生一个常规函数调用。

一如往常，名字查找先于类型检查

如前所述，声明在内层作用域的函数并不会重载声明在外层作用域的函数（参见 6.4.1 节，第 210 页）。因此，定义派生类中的函数也不会重载其基类中的成员。和其他作用域一样，如果派生类（即内层作用域）的成员与基类（即外层作用域）的某个成员同名，则派生类将在其作用域内隐藏该基类成员。即使派生类成员和基类成员的形参列表不一致，基类成员也仍然会被隐藏掉：

```
struct Base {
    int memfcn();
};
struct Derived : Base {
    int memfcn(int);            // 隐藏基类的 memfcn
};
Derived d; Base b;
b.memfcn();                     // 调用 Base::memfcn
d.memfcn(10);                   // 调用 Derived::memfcn
d.memfcn();                     // 错误：参数列表为空的 memfcn 被隐藏了
d.Base::memfcn();               // 正确：调用 Base::memfcn
```

Derived 中的 memfcn 声明隐藏了 Base 中的 memfcn 声明。在上面的代码中前两条调用语句容易理解，第一个通过 Base 对象 b 进行的调用执行基类的版本；类似的，第二个通过 d 进行的调用执行 Derived 的版本；第三条调用语句有点特殊，d.memfcn()是非法的。

620

为了解析这条调用语句，编译器首先在 Derived 中查找名字 memfcn；因为 Derived 确实定义了一个名为 memfcn 的成员，所以查找过程终止。一旦名字找到，编译器就不再继续查找了。Derived 中的 memfcn 版本需要一个 int 实参，而当前的调用语句无法提供任何实参，所以该调用语句是错误的。

虚函数与作用域

我们现在可以理解为什么基类与派生类中的虚函数必须有相同的形参列表了（参见 15.3 节，第 537 页）。假如基类与派生类的虚函数接受的实参不同，则我们就无法通过基类的引用或指针调用派生类的虚函数了。例如：

```
class Base {
public:
    virtual int fcn();
};
class D1 : public Base {
public:
    // 隐藏基类的 fcn，这个 fcn 不是虚函数
    // D1 继承了 Base::fcn()的定义
    int fcn(int);                    // 形参列表与 Base 中的 fcn 不一致
    virtual void f2();               // 是一个新的虚函数，在 Base 中不存在
};
class D2 : public D1 {
public:
    int fcn(int);                    // 是一个非虚函数，隐藏了 D1::fcn(int)
    int fcn();                       // 覆盖了 Base 的虚函数 fcn
    void f2();                       // 覆盖了 D1 的虚函数 f2
};
```

D1 的 fcn 函数并没有覆盖 Base 的虚函数 fcn，原因是它们的形参列表不同。实际上，D1 的 fcn 将隐藏 Base 的 fcn。此时拥有了两个名为 fcn 的函数：一个是 D1 从 Base 继承而来的虚函数 fcn；另一个是 D1 自己定义的接受一个 int 参数的非虚函数 fcn。

通过基类调用隐藏的虚函数

给定上面定义的这些类后，我们来看几种使用其函数的方法：

```
Base bobj; D1 d1obj; D2 d2obj;

Base *bp1 = &bobj, *bp2 = &d1obj, *bp3 = &d2obj;
bp1->fcn();                 // 虚调用，将在运行时调用 Base::fcn
bp2->fcn();                 // 虚调用，将在运行时调用 Base::fcn
bp3->fcn();                 // 虚调用，将在运行时调用 D2::fcn

D1 *d1p = &d1obj; D2 *d2p = &d2obj;
bp2->f2();                  // 错误：Base 没有名为 f2 的成员
d1p->f2();                  // 虚调用，将在运行时调用 D1::f2()
d2p->f2();                  // 虚调用，将在运行时调用 D2::f2()
```

前三条调用语句是通过基类的指针进行的，因为 fcn 是虚函数，所以编译器产生的代码将在运行时确定使用虚函数的哪个版本。判断的依据是该指针所绑定对象的真实类型。在 bp2 的例子中，实际绑定的对象是 D1 类型，而 D1 并没有覆盖那个不接受实参的 fcn，所以通过 bp2 进行的调用将在运行时解析为 Base 定义的版本。 621

接下来的三条调用语句是通过不同类型的指针进行的，每个指针分别指向继承体系中的一个类型。因为 Base 类中没有 f2()，所以第一条语句是非法的，即使当前的指针碰巧指向了一个派生类对象也无济于事。

为了完整地阐明上述问题，我们不妨再观察一些对于非虚函数 fcn(int)的调用语句：

```
Base *p1 = &d2obj; D1 *p2 = &d2obj; D2 *p3 = &d2obj;
p1->fcn(42);                    // 错误：Base 中没有接受一个 int 的 fcn
p2->fcn(42);                    // 静态绑定，调用 D1::fcn(int)
p3->fcn(42);                    // 静态绑定，调用 D2::fcn(int)
```

在上面的每条调用语句中，指针都指向了 D2 类型的对象，但是由于我们调用的是非虚函数，所以不会发生动态绑定。实际调用的函数版本由指针的静态类型决定。

覆盖重载的函数

和其他函数一样，成员函数无论是否是虚函数都能被重载。派生类可以覆盖重载函数的 0 个或多个实例。如果派生类希望所有的重载版本对于它来说都是可见的，那么它就需要覆盖所有的版本，或者一个也不覆盖。

有时一个类仅需覆盖重载集合中的一些而非全部函数，此时，如果我们不得不覆盖基类中的每一个版本的话，显然操作将极其烦琐。

一种好的解决方案是为重载的成员提供一条 using 声明语句（参见 15.5 节，第 546 页），这样我们就无须覆盖基类中的每一个重载版本了。using 声明语句指定一个名字而不指定形参列表，所以一条基类成员函数的 using 声明语句就可以把该函数的所有重载实例添加到派生类作用域中。此时，派生类只需要定义其特有的函数就可以了，而无须为继承而来的其他函数重新定义。

类内 using 声明的一般规则同样适用于重载函数的名字（参见 15.5 节，第 546 页）；基类函数的每个实例在派生类中都必须是可访问的。对派生类没有重新定义的重载版本的访问实际上是对 using 声明点的访问。

15.6 节练习

练习 15.23：假设第 550 页的 D1 类需要覆盖它继承而来的 fcn 函数，你应该如何对其进行修改？如果你修改之后 fcn 匹配了 Base 中的定义，则该节的那些调用语句将如何解析？

15.7 构造函数与拷贝控制

622

和其他类一样，位于继承体系中的类也需要控制当其对象执行一系列操作时发生什么样的行为，这些操作包括创建、拷贝、移动、赋值和销毁。如果一个类（基类或派生类）没有定义拷贝控制操作，则编译器将为它合成一个版本。当然，这个合成的版本也可以定义成被删除的函数。

15.7.1 虚析构函数

继承关系对基类拷贝控制最直接的影响是基类通常应该定义一个虚析构函数（参见 15.2.1 节，第 528 页），这样我们就能动态分配继承体系中的对象了。

如前所述，当我们 delete 一个动态分配的对象的指针时将执行析构函数（参见 13.1.3 节，第 445 页）。如果该指针指向继承体系中的某个类型，则有可能出现指针的静态类型与被删除对象的动态类型不符的情况（参见 15.2.2 节，第 530 页）。例如，如果我们 delete 一个 Quote*类型的指针，则该指针有可能实际指向了一个 Bulk_quote 类型的对象。如果这样的话，编译器就必须清楚它应该执行的是 Bulk_quote 的析构函数。和其他函数一样，我们通过在基类中将析构函数定义成虚函数以确保执行正确的析构函数版本：

```
class Quote {
public:
    // 如果我们删除的是一个指向派生类对象的基类指针，则需要虚析构函数
    virtual ~Quote() = default;        // 动态绑定析构函数
};
```

和其他虚函数一样，析构函数的虚属性也会被继承。因此，无论 Quote 的派生类使用合成的析构函数还是定义自己的析构函数，都将是虚析构函数。只要基类的析构函数是虚函数，就能确保当我们 delete 基类指针时将运行正确的析构函数版本：

```
Quote *itemP = new Quote;          // 静态类型与动态类型一致
delete itemP;                      // 调用 Quote 的析构函数
itemP = new Bulk_quote;            // 静态类型与动态类型不一致
delete itemP;                      // 调用 Bulk_quote 的析构函数
```

如果基类的析构函数不是虚函数，则 delete 一个指向派生类对象的基类指针将产生未定义的行为。

之前我们曾介绍过一条经验准则，即如果一个类需要析构函数，那么它也同样需要拷贝和赋值操作（参见 13.1.4 节，第 447 页）。基类的析构函数并不遵循上述准则，它是一个重要的例外。一个基类总是需要析构函数，而且它能将析构函数设定为虚函数。此时，该析构函数为了成为虚函数而令内容为空，我们显然无法由此推断该基类还需要赋值运算符或拷贝构造函数。

623 虚析构函数将阻止合成移动操作

基类需要一个虚析构函数这一事实还会对基类和派生类的定义产生另外一个间接的影响：如果一个类定义了析构函数，即使它通过=default 的形式使用了合成的版本，编译器也不会为这个类合成移动操作（参见 13.6.2 节，第 475 页）。

15.7.1 节练习

练习 15.24：哪种类需要虚析构函数？虚析构函数必须执行什么样的操作？

15.7.2 合成拷贝控制与继承

基类或派生类的合成拷贝控制成员的行为与其他合成的构造函数、赋值运算符或析构函数类似：它们对类本身的成员依次进行初始化、赋值或销毁的操作。此外，这些合成的成员还负责使用直接基类中对应的操作对一个对象的直接基类部分进行初始化、赋值或销

毁的操作。例如，

- 合成的 Bulk_quote 默认构造函数运行 Disc_quote 的默认构造函数，后者又运行 Quote 的默认构造函数。
- Quote 的默认构造函数将 bookNo 成员默认初始化为空字符串，同时使用类内初始值将 price 初始化为 0。
- Quote 的构造函数完成后，继续执行 Disc_quote 的构造函数，它使用类内初始值初始化 qty 和 discount。
- Disc_quote 的构造函数完成后，继续执行 Bulk_quote 的构造函数，但是它什么具体工作也不做。

类似的，合成的 Bulk_quote 拷贝构造函数使用（合成的）Disc_quote 拷贝构造函数，后者又使用（合成的）Quote 拷贝构造函数。其中，Quote 拷贝构造函数拷贝 bookNo 和 price 成员；Disc_quote 拷贝构造函数拷贝 qty 和 discount 成员。

值得注意的是，无论基类成员是合成的版本（如 Quote 继承体系的例子）还是自定义的版本都没有太大影响。唯一的要求是相应的成员应该可访问（参见 15.5 节，第 542 页）并且不是一个被删除的函数。

在我们的 Quote 继承体系中，所有类都使用合成的析构函数。其中，派生类隐式地使用而基类通过将其虚析构函数定义成=default 而显式地使用。一如既往，合成的析构函数体是空的，其隐式的析构部分负责销毁类的成员（参见 13.1.3 节，第 444 页）。对于派生类的析构函数来说，它除了销毁派生类自己的成员外，还负责销毁派生类的直接基类；该直接基类又销毁它自己的直接基类，以此类推直至继承链的顶端。 C++ 11

如前所述，Quote 因为定义了析构函数而不能拥有合成的移动操作，因此当我们移动 Quote 对象时实际使用的是合成的拷贝操作（参见 13.6.2 节，第 477 页）。如我们即将看到的那样，Quote 没有移动操作意味着它的派生类也没有。 624

派生类中删除的拷贝控制与基类的关系

就像其他任何类的情况一样，基类或派生类也能出于同样的原因将其合成的默认构造函数或者任何一个拷贝控制成员定义成被删除的函数（参见 13.1.6 节，第 450 页和 13.6.2 节，第 475 页）。此外，某些定义基类的方式也可能导致有的派生类成员成为被删除的函数： C++ 11

- 如果基类中的默认构造函数、拷贝构造函数、拷贝赋值运算符或析构函数是被删除的函数或者不可访问（参见 15.5 节，第 543 页），则派生类中对应的成员将是被删除的，原因是编译器不能使用基类成员来执行派生类对象基类部分的构造、赋值或销毁操作。
- 如果在基类中有一个不可访问或删除掉的析构函数，则派生类中合成的默认和拷贝构造函数将是被删除的，因为编译器无法销毁派生类对象的基类部分。
- 和过去一样，编译器将不会合成一个删除掉的移动操作。当我们使用=default 请求一个移动操作时，如果基类中的对应操作是删除的或不可访问的，那么派生类中该函数将是被删除的，原因是派生类对象的基类部分不可移动。同样，如果基类的析构函数是删除的或不可访问的，则派生类的移动构造函数也将是被删除的。

举个例子，对于下面的基类 B 来说：

```
class B {
public:
    B();
```

```
    B(const B&) = delete;
    // 其他成员，不含有移动构造函数
};
class D : public B {
    // 没有声明任何构造函数
};
D d;                    // 正确：D 的合成默认构造函数使用 B 的默认构造函数
D d2(d);                // 错误：D 的合成拷贝构造函数是被删除的
D d3(std::move(d));     // 错误：隐式地使用 D 的被删除的拷贝构造函数
```

基类 B 含有一个可访问的默认构造函数和一个显式删除的拷贝构造函数。因为我们定义了拷贝构造函数，所以编译器将不会为 B 合成一个移动构造函数（参见 13.6.2 节，第 475 页）。因此，我们既不能移动也不能拷贝 B 的对象。如果 B 的派生类希望它自己的对象能被移动和拷贝，则派生类需要自定义相应版本的构造函数。当然，在这一过程中派生类还必须考虑如何移动或拷贝其基类部分的成员。在实际编程过程中，如果在基类中没有默认、拷贝或移动构造函数，则一般情况下派生类也不会定义相应的操作。

625 **移动操作与继承**

如前所述，大多数基类都会定义一个虚析构函数。因此在默认情况下，基类通常不含有合成的移动操作，而且在它的派生类中也没有合成的移动操作。

因为基类缺少移动操作会阻止派生类拥有自己的合成移动操作，所以当我们确实需要执行移动操作时应该首先在基类中进行定义。我们的 Quote 可以使用合成的版本，不过前提是 Quote 必须显式地定义这些成员。一旦 Quote 定义了自己的移动操作，那么它必须同时显式地定义拷贝操作（参见 13.6.2 节，第 476 页）：

```
class Quote {
public:
    Quote() = default;                          // 对成员依次进行默认初始化
    Quote(const Quote&) = default;              // 对成员依次拷贝
    Quote(Quote&&) = default;                   // 对成员依次拷贝
    Quote& operator=(const Quote&) = default;   // 拷贝赋值
    Quote& operator=(Quote&&) = default;        // 移动赋值
    virtual ~Quote() = default;
    // 其他成员与之前的版本一致
};
```

通过上面的定义，我们就能对 Quote 的对象逐成员地分别进行拷贝、移动、赋值和销毁操作了。而且除非 Quote 的派生类中含有排斥移动的成员，否则它将自动获得合成的移动操作。

15.7.2 节练习

练习 15.25：我们为什么为 Disc_quote 定义一个默认构造函数？如果去除掉该构造函数的话会对 Bulk_quote 的行为产生什么影响？

15.7.3 派生类的拷贝控制成员

如我们在 15.2.2 节（第 531 页）介绍过的，派生类构造函数在其初始化阶段中不但要初始化派生类自己的成员，还负责初始化派生类对象的基类部分。因此，派生类的拷贝和

移动构造函数在拷贝和移动自有成员的同时，也要拷贝和移动基类部分的成员。类似的，派生类赋值运算符也必须为其基类部分的成员赋值。

和构造函数及赋值运算符不同的是，析构函数只负责销毁派生类自己分配的资源。如前所述，对象的成员是被隐式销毁的（参见 13.1.3 节，第 445 页）；类似的，派生类对象的基类部分也是自动销毁的。

当派生类定义了拷贝或移动操作时，该操作负责拷贝或移动包括基类部分成员在内的整个对象。

626

定义派生类的拷贝或移动构造函数

当为派生类定义拷贝或移动构造函数时（参见 13.1.1 节，第 440 页和 13.6.2 节，第 473 页），我们通常使用对应的基类构造函数初始化对象的基类部分：

```
class Base { /* ...*/ };
class D: public Base {
public:
    // 默认情况下，基类的默认构造函数初始化对象的基类部分
    // 要想使用拷贝或移动构造函数，我们必须在构造函数初始值列表中
    // 显式地调用该构造函数
    D(const D& d): Base(d)                    // 拷贝基类成员
                  /* D的成员的初始值 */ { /* ... */ }
    D(D&& d): Base(std::move(d))        // 移动基类成员
                  /* D的成员的初始值 */ { /* ... */ }
};
```

初始值 Base(d) 将一个 D 对象传递给基类构造函数。尽管从道理上来说，Base 可以包含一个参数类型为 D 的构造函数，但是在实际编程过程中通常不会这么做。相反，Base(d) 一般会匹配 Base 的拷贝构造函数。D 类型的对象 d 将被绑定到该构造函数的 Base&形参上。Base 的拷贝构造函数负责将 d 的基类部分拷贝给要创建的对象。假如我们没有提供基类的初始值的话：

```
// D的这个拷贝构造函数很可能是不正确的定义
// 基类部分被默认初始化，而非拷贝
D(const D& d) /* 成员初始值，但是没有提供基类初始值*/
    { /* ...*/ }
```

在上面的例子中，Base 的默认构造函数将被用来初始化 D 对象的基类部分。假定 D 的构造函数从 d 中拷贝了派生类成员，则这个新构建的对象的配置将非常奇怪：它的 Base 成员被赋予了默认值，而 D 成员的值则是从其他对象拷贝得来的。

在默认情况下，基类默认构造函数初始化派生类对象的基类部分。如果我们想拷贝（或移动）基类部分，则必须在派生类的构造函数初始值列表中显式地使用基类的拷贝（或移动）构造函数。

派生类赋值运算符

与拷贝和移动构造函数一样，派生类的赋值运算符（参见 13.1.2 节，第 443 页和 13.6.2 节，第 474 页）也必须显式地为其基类部分赋值：

```
// Base::operator=(const Base&)不会被自动调用
```

627

```
D &D::operator=(const D &rhs)
{
    Base::operator=(rhs); // 为基类部分赋值
    // 按照过去的方式为派生类的成员赋值
    // 酌情处理自赋值及释放已有资源等情况
    return *this;
}
```

上面的运算符首先显式地调用基类赋值运算符，令其为派生类对象的基类部分赋值。基类的运算符（应该可以）正确地处理自赋值的情况，如果赋值命令是正确的，则基类运算符将释放掉其左侧运算对象的基类部分的旧值，然后利用 rhs 为其赋一个新值。随后，我们继续进行其他为派生类成员赋值的工作。

值得注意的是，无论基类的构造函数或赋值运算符是自定义的版本还是合成的版本，派生类的对应操作都能使用它们。例如，对于 Base::operator=的调用语句将执行 Base 的拷贝赋值运算符，至于该运算符是由 Base 显式定义的还是由编译器合成的无关紧要。

派生类析构函数

如前所述，在析构函数体执行完成后，对象的成员会被隐式销毁（参见 13.1.3 节，第 445 页）。类似的，对象的基类部分也是隐式销毁的。因此，和构造函数及赋值运算符不同的是，派生类析构函数只负责销毁由派生类自己分配的资源：

```
class D: public Base {
public:
    // Base::~Base 被自动调用执行
    ~D() { /* 该处由用户定义清除派生类成员的操作 */ }
};
```

对象销毁的顺序正好与其创建的顺序相反：派生类析构函数首先执行，然后是基类的析构函数，以此类推，沿着继承体系的反方向直至最后。

在构造函数和析构函数中调用虚函数

如我们所知，派生类对象的基类部分将首先被构建。当执行基类的构造函数时，该对象的派生类部分是未被初始化的状态。类似的，销毁派生类对象的次序正好相反，因此当执行基类的析构函数时，派生类部分已经被销毁掉了。由此可知，当我们执行上述基类成员的时候，该对象处于未完成的状态。

为了能够正确地处理这种未完成状态，编译器认为对象的类型在构造或析构的过程中仿佛发生了改变一样。也就是说，当我们构建一个对象时，需要把对象的类和构造函数的类看作是同一个；对虚函数的调用绑定正好符合这种把对象的类和构造函数的类看成同一个的要求；对于析构函数也是同样的道理。上述的绑定不但对直接调用虚函数有效，对间接调用也是有效的，这里的间接调用是指通过构造函数（或析构函数）调用另一个函数。

628

为了理解上述行为，不妨考虑当基类构造函数调用虚函数的派生类版本时会发生什么情况。这个虚函数可能会访问派生类的成员，毕竟，如果它不需要访问派生类成员的话，则派生类直接使用基类的虚函数版本就可以了。然而，当执行基类构造函数时，它要用到的派生类成员尚未初始化，如果我们允许这样的访问，则程序很可能会崩溃。

如果构造函数或析构函数调用了某个虚函数，则我们应该执行与构造函数或析构函数所属类型相对应的虚函数版本。

15.7.3 节练习

练习 15.26：定义 Quote 和 Bulk_quote 的拷贝控制成员，令其与合成的版本行为一致。为这些成员以及其他构造函数添加打印状态的语句，使得我们能够知道正在运行哪个程序。使用这些类编写程序，预测程序将创建和销毁哪些对象。重复实验，不断比较你的预测和实际输出结果是否相同，直到预测完全准确再结束。

15.7.4 继承的构造函数

在 C++11 新标准中，派生类能够重用其直接基类定义的构造函数。尽管如我们所知，这些构造函数并非以常规的方式继承而来，但是为了方便，我们不妨姑且称其为“继承”的。一个类只初始化它的直接基类，出于同样的原因，一个类也只继承其直接基类的构造函数。类不能继承默认、拷贝和移动构造函数。如果派生类没有直接定义这些构造函数，则编译器将为派生类合成它们。

C++ 11

派生类继承基类构造函数的方式是提供一条注明了（直接）基类名的 using 声明语句。举个例子，我们可以重新定义 Bulk_quote 类（参见 15.4 节，第 541 页），令其继承 Disc_quote 类的构造函数：

```
class Bulk_quote : public Disc_quote {
public:
    using Disc_quote::Disc_quote; // 继承 Disc_quote 的构造函数
    double net_price(std::size_t) const;
};
```

通常情况下，using 声明语句只是令某个名字在当前作用域内可见。而当作用于构造函数时，using 声明语句将令编译器产生代码。对于基类的每个构造函数，编译器都生成一个与之对应的派生类构造函数。换句话说，对于基类的每个构造函数，编译器都在派生类中生成一个形参列表完全相同的构造函数。

这些编译器生成的构造函数形如：

629

derived(*parms*) : *base*(*args*) { }

其中，*derived* 是派生类的名字，*base* 是基类的名字，*parms* 是构造函数的形参列表，*args* 将派生类构造函数的形参传递给基类的构造函数。在我们的 Bulk_quote 类中，继承的构造函数等价于：

```
Bulk_quote(const std::string& book, double price,
           std::size_t qty, double disc):
      Disc_quote(book, price, qty, disc) { }
```

如果派生类含有自己的数据成员，则这些成员将被默认初始化（参见 7.1.4 节，第 238 页）。

继承的构造函数的特点

和普通成员的 using 声明不一样，一个构造函数的 using 声明不会改变该构造函数的访问级别。例如，不管 using 声明出现在哪儿，基类的私有构造函数在派生类中还是一个私有构造函数；受保护的构造函数和公有构造函数也是同样的规则。

而且，一个 using 声明语句不能指定 explicit 或 constexpr。如果基类的构造函数是 explicit（参见 7.5.4 节，第 265 页）或者 constexpr（参见 7.5.6 节，第 267

页），则继承的构造函数也拥有相同的属性。

当一个基类构造函数含有默认实参（参见 6.5.1 节，第 211 页）时，这些实参并不会被继承。相反，派生类将获得多个继承的构造函数，其中每个构造函数分别省略掉一个含有默认实参的形参。例如，如果基类有一个接受两个形参的构造函数，其中第二个形参含有默认实参，则派生类将获得两个构造函数：一个构造函数接受两个形参（没有默认实参），另一个构造函数只接受一个形参，它对应于基类中最左侧的没有默认值的那个形参。

如果基类含有几个构造函数，则除了两个例外情况，大多数时候派生类会继承所有这些构造函数。第一个例外是派生类可以继承一部分构造函数，而为其他构造函数定义自己的版本。如果派生类定义的构造函数与基类的构造函数具有相同的参数列表，则该构造函数将不会被继承。定义在派生类中的构造函数将替换继承而来的构造函数。

第二个例外是默认、拷贝和移动构造函数不会被继承。这些构造函数按照正常规则被合成。继承的构造函数不会被作为用户定义的构造函数来使用，因此，如果一个类只含有继承的构造函数，则它也将拥有一个合成的默认构造函数。

15.7.4 节练习

练习 15.27：重新定义你的 Bulk_quote 类，令其继承构造函数。

15.8 容器与继承

630

当我们使用容器存放继承体系中的对象时，通常必须采取间接存储的方式。因为不允许在容器中保存不同类型的元素，所以我们不能把具有继承关系的多种类型的对象直接存放在容器当中。

举个例子，假定我们想定义一个 vector，令其保存用户准备购买的几种书籍。显然我们不应该用 vector 保存 Bulk_quote 对象。因为我们不能将 Quote 对象转换成 Bulk_quote（参见 15.2.3 节，第 534 页），所以我们将无法把 Quote 对象放置在该 vector 中。

其实，我们也不应该使用 vector 保存 Quote 对象。此时，虽然我们可以把 Bulk_quote 对象放置在容器中，但是这些对象再也不是 Bulk_quote 对象了：

```
vector<Quote> basket;
basket.push_back(Quote("0-201-82470-1", 50));
// 正确：但是只能把对象的 Quote 部分拷贝给 basket
basket.push_back(Bulk_quote("0-201-54848-8", 50, 10, .25));
// 调用 Quote 定义的版本，打印 750，即 15 * $50
cout << basket.back().net_price(15) << endl;
```

basket 的元素是 Quote 对象，因此当我们向该 vector 中添加一个 Bulk_quote 对象时，它的派生类部分将被忽略掉（参见 15.2.3 节，第 535 页）。

当派生类对象被赋值给基类对象时，其中的派生类部分将被“切掉”，因此容器和存在继承关系的类型无法兼容。

在容器中放置（智能）指针而非对象

当我们希望在容器中存放具有继承关系的对象时，我们实际上存放的通常是基类的指针（更好的选择是智能指针（参见 12.1 节，第 400 页））。和往常一样，这些指针所指对象的动态类型可能是基类类型，也可能是派生类类型：

```
vector<shared_ptr<Quote>> basket;

basket.push_back(make_shared<Quote>("0-201-82470-1", 50));
basket.push_back(
    make_shared<Bulk_quote>("0-201-54848-8", 50, 10, .25));
// 调用 Quote 定义的版本；打印 562.5，即在 15*&50 中扣除掉折扣金额
cout << basket.back()->net_price(15) << endl;
```

因为 basket 存放着 shared_ptr，所以我们必须解引用 basket.back()的返回值以获得运行 net_price 的对象。我们通过在 net_price 的调用中使用->以达到这个目的。如我们所知，实际调用的 net_price 版本依赖于指针所指对象的动态类型。

值得注意的是，我们将 basket 定义成 shared_prt<Quote>，但是在第二个 push_back 中传入的是一个 Bulk_quote 对象的 shared_ptr。正如我们可以将一个派生类的普通指针转换成基类指针一样（参见 15.2.2 节，第 530 页），我们也能把一个派生类的智能指针转换成基类的智能指针。在此例中，make_shared<Bulk_quote>返回一个 shared_ptr<Bulk_quote>对象，当我们调用 push_back 时该对象被转换成 shared_ptr<Quote>。因此尽管在形式上有所差别，但实际上 basket 的所有元素的类型都是相同的。 631

15.8 节练习

练习 15.28：定义一个存放 Quote 对象的 vector，将 Bulk_quote 对象传入其中。计算 vector 中所有元素总的 net_price。

练习 15.29：再运行一次你的程序，这次传入 Quote 对象的 shared_ptr。如果这次计算出的总额与之前的程序不一致，解释为什么；如果一致，也请说明原因。

15.8.1 编写 Basket 类

对于 C++面向对象的编程来说，一个悖论是我们无法直接使用对象进行面向对象编程。相反，我们必须使用指针和引用。因为指针会增加程序的复杂性，所以我们经常定义一些辅助的类来处理这种复杂情况。首先，我们定义一个表示购物篮的类：

```
class Basket {
public:
    // Basket 使用合成的默认构造函数和拷贝控制成员
    void add_item(const std::shared_ptr<Quote> &sale)
        { items.insert(sale); }
    // 打印每本书的总价和购物篮中所有书的总价
    double total_receipt(std::ostream&) const;
private:
    // 该函数用于比较 shared_ptr，multiset 成员会用到它
    static bool compare(const std::shared_ptr<Quote> &lhs,
                        const std::shared_ptr<Quote> &rhs)
    { return lhs->isbn() < rhs->isbn(); }
    // multiset 保存多个报价，按照 compare 成员排序
```

```
    std::multiset<std::shared_ptr<Quote>, decltype(compare)*>
                    items{compare};
};
```

我们的类使用一个 multiset（参见 11.2.1 节，第 377 页）来存放交易信息，这样我们就能保存同一本书的多条交易记录，而且对于一本给定的书籍，它的所有交易信息都保存在一起（参见 11.2.2 节，第 378 页）。

multiset 的元素是 shared_ptr。因为 shared_ptr 没有定义小于运算符，所以为了对元素排序我们必须提供自己的比较运算符（参见 11.2.2 节，第 378 页）。在此例中，我们定义了一个名为 compare 的私有静态成员，该成员负责比较 shared_ptr 所指的对象的 isbn。我们初始化 multiset，通过类内初始值调用比较函数（参见 7.3.1 节，第 246 页）：

632

```
// multiset 保存多个报价，按照 compare 成员排序
std::multiset<std::shared_ptr<Quote>, decltype(compare)*>
              items{compare};
```

这个声明看起来不太容易理解，但是从左向右读的话，我们就能明白它其实是定义了一个指向 Quote 对象的 shared_ptr 的 multiset。这个 multiset 将使用一个与 compare 成员类型相同的函数来对其中的元素进行排序。multiset 成员的名字是 items，我们初始化 items 并令其使用我们的 compare 函数。

定义 Basket 的成员

Basket 类只定义两个操作。第一个成员是我们在类的内部定义的 add_item 成员，该成员接受一个指向动态分配的 Quote 的 shared_ptr，然后将这个 shared_ptr 放置在 multiset 中。第二个成员的名字是 total_receipt，它负责将购物篮的内容逐项打印成清单，然后返回购物篮中所有物品的总价格：

```
double Basket::total_receipt(ostream &os) const
{
    double sum = 0.0;                    // 保存实时计算出的总价格
    // iter 指向 ISBN 相同的一批元素中的第一个
    // upper_bound 返回一个迭代器，该迭代器指向这批元素的尾后位置
    for (auto iter = items.cbegin();
             iter != items.cend();
             iter = items.upper_bound(*iter)) {
        // 我们知道在当前的 Basket 中至少有一个该关键字的元素
        // 打印该书籍对应的项目
        sum += print_total(os, **iter, items.count(*iter));
    }
    os << "Total Sale: " << sum << endl;  // 打印最终的总价格
    return sum;
}
```

我们的 for 循环首先定义并初始化 iter，令其指向 multiset 的第一个元素。条件部分检查 iter 是否等于 items.cend()：如果相等，表明我们已经处理完了所有购买记录，接下来应该跳出 for 循环；否则，如果不相等，则继续处理下一本书籍。

比较有趣的是，for 循环中的“递增”表达式。与通常的循环语句依次读取每个元素不同，我们直接令 iter 指向下一个关键字，调用 upper_bound 函数可以令我们跳过与当前关键字相同的所有元素（参见 11.3.5 节，第 390 页）。对于 upper_bound 函数来说，它返回的是一个迭代器，该迭代器指向所有与 iter 关键字相等的元素中最后一个元素的

下一位置。因此，我们得到的迭代器或者指向集合的末尾，或者指向下一本书籍。

在 for 循环内部，我们通过调用 print_total（参见 15.1 节，第 527 页）来打印购物篮中每本书籍的细节：

```
sum += print_total(os, **iter, items.count(*iter));
```

print_total 的实参包括一个用于写入数据的 ostream、一个待处理的 Quote 对象和一个计数值。当我们解引用 iter 后将得到一个指向准备打印的对象的 shared_ptr。为了得到这个对象，必须解引用该 shared_ptr。因此，**iter 是一个 Quote 对象（或者 Quote 的派生类的对象）。我们使用 multiset 的 count 成员（参见 11.3.5 节，第 388 页）来统计在 multiset 中有多少元素的键值相同（即 ISBN 相同）。

633

如我们所知，print_total 调用了虚函数 net_price，因此最终的计算结果依赖于**iter 的动态类型。print_total 函数打印并返回给定书籍的总价格，我们把这个结果添加到 sum 当中，最后当循环结束后打印 sum。

隐藏指针

Basket 的用户仍然必须处理动态内存，原因是 add_item 需要接受一个 shared_ptr 参数。因此，用户不得不按照如下形式编写代码：

```
Basket bsk;
bsk.add_item(make_shared<Quote>("123", 45));
bsk.add_item(make_shared<Bulk_quote>("345", 45, 3, .15));
```

我们的下一步是重新定义 add_item，使得它接受一个 Quote 对象而非 shared_ptr。新版本的 add_item 将负责处理内存分配，这样它的用户就不必再受困于此了。我们将定义两个版本，一个拷贝它给定的对象，另一个则采取移动操作（参见 13.6.3 节，第 481 页）：

```
void add_item(const Quote& sale);          // 拷贝给定的对象
void add_item(Quote&& sale);               // 移动给定的对象
```

唯一的问题是 add_item 不知道要分配的类型。当 add_item 进行内存分配时，它将拷贝（或移动）它的 sale 参数。在某处可能会有一条如下形式的 new 表达式：

```
new Quote(sale)
```

不幸的是，这条表达式所做的工作可能是不正确的：new 为我们请求的类型分配内存，因此这条表达式将分配一个 Quote 类型的对象并且拷贝 sale 的 Quote 部分。然而，sale 实际指向的可能是 Bulk_quote 对象，此时，该对象将被迫切掉一部分。

模拟虚拟拷贝

为了解决上述问题，我们给 Quote 类添加一个虚函数，该函数将申请一份当前对象的拷贝。

```
class Quote {
public:
    // 该虚函数返回当前对象的一份动态分配的拷贝
    // 这些成员使用的引用限定符参见 13.6.3 节（第 483 页）
    virtual Quote* clone() const & {return new Quote(*this);}
    virtual Quote* clone() &&
                        {return new Quote(std::move(*this));}
    // 其他成员与之前的版本一致
};
```

634

```
class Bulk_quote : public Quote {
    Bulk_quote* clone() const & {return new Bulk_quote(*this);}
    Bulk_quote* clone() &&
                    {return new Bulk_quote(std::move(*this));}
    // 其他成员与之前的版本一致
};
```

因为我们拥有 add_item 的拷贝和移动版本，所以我们分别定义 clone 的左值和右值版本(参见 13.6.3 节，第 483 页)。每个 clone 函数分配当前类型的一个新对象，其中，const 左值引用成员将它自己拷贝给新分配的对象；右值引用成员则将自己移动到新数据中。

我们可以使用 clone 很容易地写出新版本的 add_item：

```
class Basket {
public:
    void add_item(const Quote& sale)      // 拷贝给定的对象
      { items.insert(std::shared_ptr<Quote>(sale.clone())); }
    void add_item(Quote&& sale)           // 移动给定的对象
      { items.insert(
          std::shared_ptr<Quote>(std::move(sale).clone())); }
    // 其他成员与之前的版本一致
};
```

和 add_item 本身一样，clone 函数也根据作用于左值还是右值而分为不同的重载版本。在此例中，第一个 add_item 函数调用 clone 的 const 左值版本，第二个函数调用 clone 的右值引用版本。在右值版本中，尽管 sale 的类型是右值引用类型，但实际上 sale 本身（和任何其他变量一样）是个左值（参见 13.6.1 节，第 471 页）。因此，我们调用 move 把一个右值引用绑定到 sale 上。

我们的 clone 函数也是一个虚函数。sale 的动态类型(通常)决定了到底运行 Quote 的函数还是 Bulk_quote 的函数。无论我们是拷贝还是移动数据，clone 都返回一个新分配对象的指针，该对象与 clone 所属的类型一致。我们把一个 shared_ptr 绑定到这个对象上，然后调用 insert 将这个新分配的对象添加到 items 中。注意，因为 shared_ptr 支持派生类向基类的类型转换（参见 15.2.2 节，第 530 页），所以我们能把 shared_ptr<Quote>绑定到 Bulk_quote*上。

15.8.1 节练习

练习 15.30：编写你自己的 Basket 类，用它计算上一个练习中交易记录的总价格。

15.9 文本查询程序再探

接下来，我们扩展 12.3 节（第 430 页）的文本查询程序，用它作为说明继承的最后一个例子。在上一版的程序中，我们可以查询在文件中某个指定单词的出现情况。我们将在本节扩展该程序使其支持更多更复杂的查询操作。在后面的例子中，我们将针对下面这个小故事展开查询：

635

```
Alice Emma has long flowing red hair.
Her Daddy says when the wind blows
through her hair, it looks almost alive,
like a fiery bird in flight.
```

```
A beautiful fiery bird, he tells her,
magical but untamed.
"Daddy, shush, there is no such thing,"
she tells him, at the same time wanting
him to tell her more.
Shyly, she asks, "I mean, Daddy, is there?"
```

我们的系统将支持如下查询形式。

- 单词查询，用于得到匹配某个给定 string 的所有行：

```
Executing Query for: Daddy
Daddy occurs 3 times
(line 2) Her Daddy says when the wind blows
(line 7) "Daddy, shush, there is no such thing,"
(line 10) Shyly, she asks, "I mean, Daddy, is there?"
```

- 逻辑非查询，使用~运算符得到不匹配查询条件的所有行：

```
Executing Query for: ~(Alice)
~(Alice) occurs 9 times
(line 2) Her Daddy says when the wind blows
(line 3) through her hair, it looks almost alive,
(line 4) like a fiery bird in flight.
...
```

- 逻辑或查询，使用 | 运算符返回匹配两个条件中任意一个的行：

```
Executing Query for: (hair | Alice)
(hair | Alice) occurs 2 times
(line 1) Alice Emma has long flowing red hair.
(line 3) through her hair, it looks almost alive,
```

- 逻辑与查询，使用&运算符返回匹配全部两个条件的行：

```
Executing query for: (hair & Alice)
(hair & Alice) occurs 1 time
(line 1) Alice Emma has long flowing red hair.
```

此外，我们还希望能够混合使用这些运算符，比如：

```
fiery & bird | wind
```

在类似这样的例子中，我们将使用 C++通用的优先级规则（参见 4.1.2 节，第 121 页）对复杂表达式求值。因此，这条查询语句所得行应该是如下二者之一：在该行中或者 fiery 和 bird 同时出现，或者出现了 wind：

636

```
Executing Query for: ((fiery & bird) | wind)
((fiery & bird) | wind) occurs 3 times
(line 2) Her Daddy says when the wind blows
(line 4) like a fiery bird in flight.
(line 5) A beautiful fiery bird, he tells her,
```

在输出内容中首先是那条查询语句，我们使用圆括号来表示查询被解释和执行的次序。与之前实现的版本一样，接下来系统将按照查询结果中行号的升序显示结果并且每一行只显示一次。

15.9.1 面向对象的解决方案

我们可能会认为使用 12.3.2 节（第 432 页）的 TextQuery 类来表示单词查询，然后

从该类中派生出其他查询是一种可行的方案。

然而，这样的设计实际上存在缺陷。为了理解其中的原因，我们不妨考虑逻辑非查询。单词查询查找一个指定的单词，为了让逻辑非查询按照单词查询的方式执行，我们将不得不定义逻辑非查询所要查找的单词。但是在一般情况下，我们无法得到这样的单词。相反，一个逻辑非查询中含有一个结果值需要取反的查询语句（单词查询或任何其他查询）；类似的，一个逻辑与查询和一个逻辑或查询各包含两个结果值需要合并的查询语句。

由上述观察结果可知，我们应该将几种不同的查询建模成相互独立的类，这些类共享一个公共基类：

```
WordQuery        // Daddy
NotQuery         // ~Alice
OrQuery          // hair | Alice
AndQuery         // hair & Alice
```

这些类将只包含两个操作：

- eval，接受一个 TextQuery 对象并返回一个 QueryResult，eval 函数使用给定的 TextQuery 对象查找与之匹配的行。
- rep，返回基础查询的 string 表示形式，eval 函数使用 rep 创建一个表示匹配结果的 QueryResult，输出运算符使用 rep 打印查询表达式。

637

关键概念：继承与组合

继承体系的设计本身是一个非常复杂的问题，已经超出了本书的范围。然而，有一条设计准则非常重要也非常基础，每个程序员都应该熟悉它。

当我们令一个类公有地继承另一个类时，派生类应当反映与基类的“是一种（Is A）”关系。在设计良好的类体系当中，公有派生类的对象应该可以用在任何需要基类对象的地方。

类型之间的另一种常见关系是“有一个（Has A）”关系，具有这种关系的类暗含成员的意思。

在我们的书店示例中，基类表示的是按规定价格销售的书籍的报价。Bulk_quote“是一种”报价结果，只不过它使用的价格策略不同。我们的书店类都“有一个”价格成员和 ISBN 成员。

抽象基类

如我们所知，在这四种查询之间并不存在彼此的继承关系，从概念上来说它们互为兄弟。因为所有这些类都共享同一个接口，所以我们需要定义一个抽象基类（参见 15.4 节，第 541 页）来表示该接口。我们将所需的抽象基类命名为 Query_base，以此来表示它的角色是整个查询继承体系的根节点。

我们的 Query_base 类将把 eval 和 rep 定义成纯虚函数（参见 15.4 节，第 541 页），其他代表某种特定查询类型的类必须覆盖这两个函数。我们将从 Query_base 直接派生出 WordQuery 和 NotQuery。AndQuery 和 OrQuery 都具有系统中其他类所不具备的一个特殊属性：它们各自包含两个运算对象。为了对这种属性建模，我们定义另外一个名为 BinaryQuery 的抽象基类，该抽象基类用于表示含有两个运算对象的查询。AndQuery 和 OrQuery 继承自 BinaryQuery，而 BinaryQuery 继承自 Query_base。由这些分

析我们将得到如图 15.2 所示的类设计结果：

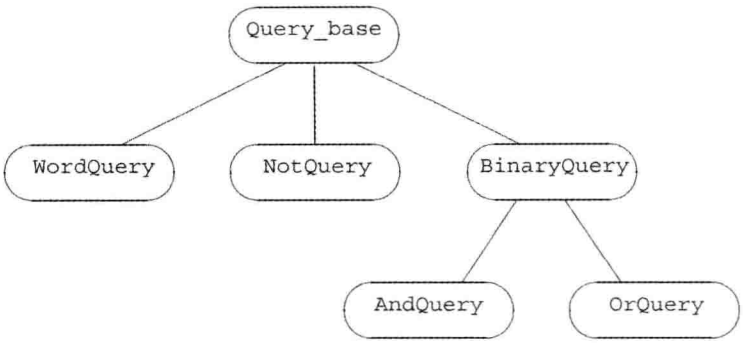


图 15.2：Query_base 继承体系

将层次关系隐藏于接口类中

我们的程序将致力于计算查询结果，而非仅仅构建查询的体系。为了使程序能正常运行，我们必须首先创建查询命令，最简单的办法是编写 C++表达式。例如，可以编写下面的代码来生成之前描述的复合查询：

```
Query q = Query("fiery") & Query("bird") | Query("wind");
```

如上所述，其隐含的意思是用户层代码将不会直接使用这些继承的类；相反，我们将定义一个名为 Query 的接口类，由它负责隐藏整个继承体系。Query 类将保存一个 Query_base 指针，该指针绑定到 Query_base 的派生类对象上。Query 类与 Query_base 类提供的操作是相同的：eval 用于求查询的结果，rep 用于生成查询的 string 版本，同时 Query 也会定义一个重载的输出运算符用于显示查询。 638

用户将通过 Query 对象的操作间接地创建并处理 Query_base 对象。我们定义 Query 对象的三个重载运算符以及一个接受 string 参数的 Query 构造函数，这些函数动态分配一个新的 Query_base 派生类的对象：

- &运算符生成一个绑定到新的 AndQuery 对象上的 Query 对象；
- |运算符生成一个绑定到新的 OrQuery 对象上的 Query 对象；
- ~运算符生成一个绑定到新的 NotQuery 对象上的 Query 对象；
- 接受 string 参数的 Query 构造函数生成一个新的 WordQuery 对象。

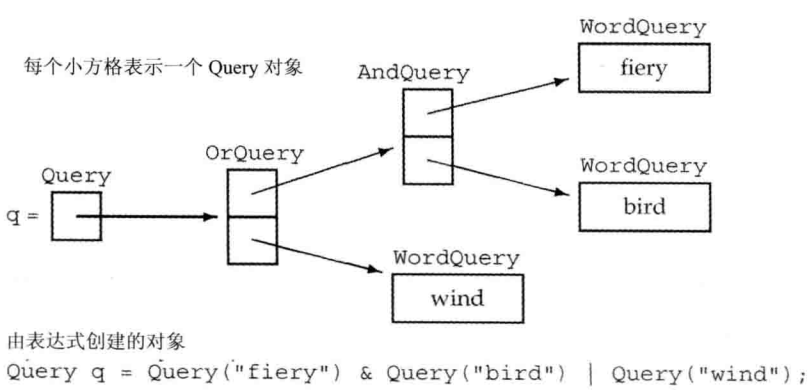


图 15.3：使用 Query 表达式创建的对象

理解这些类的工作机理

在这个应用程序中，很大一部分工作是构建代表用户查询的对象，对于读者来说认识到这一点非常重要。例如，像上面这样的表达式将生成如图 15.3 所示的一系列相关对象的集合。

一旦对象树构建完成后，对某一条查询语句的求值（或生成表示形式的）过程基本上就转换为沿着箭头方向依次对每个对象求值（或显示）的过程（由编译器为我们组织管理）。例如，如果我们对 q（即树的根节点）调用 eval 函数，则该调用语句将令 q 所指的 OrQuery 对象 eval 它自己。对该 OrQuery 求值实际上是对它的两个运算对象执行 eval 操作：一个运算对象是 AndQuery，另一个是查找单词 wind 的 WordQuery。接下来，对 AndQuery 求值转化为对它的两个 WordQuery 求值，分别生成单词 fiery 和 bird 的查询结果。

639

对于面向对象编程的新手来说，要想理解一个程序，最困难的部分往往是理解程序的设计思路。一旦你掌握了程序的设计思路，接下来的实现也就水到渠成了。为了帮助读者理解程序设计的过程，我们在表 15.1 中整理了之前那个例子用到的类，并对其进行了简要的描述。

640

表 15.1：概述：Query 程序设计

Query 程序接口类和操作	
TextQuery	该类读入给定的文件并构建一个查找图。这个类包含一个 query 操作，它接受一个 string 实参，返回一个 QueryResult 对象；该 QueryResult 对象表示 string 出现的行（12.3.2 节，第 432 页）
QueryResult	该类保存一个 query 操作的结果（12.3.2 节，第 433 页）
Query	是一个接口类，指向 Query_base 派生类的对象
Query q(s)	将 Query 对象 q 绑定到一个存放着 string s 的新 WordQuery 对象上
q1 & q2	返回一个 Query 对象，该 Query 绑定到一个存放 q1 和 q2 的新 AndQuery 对象上
q1 | q2	返回一个 Query 对象，该 Query 绑定到一个存放 q1 和 q2 的新 OrQuery 对象上
~q	返回一个 Query 对象，该 Query 绑定到一个存放 q 的新 NotQuery 对象上
Query 程序实现类	
Query_base	查询类的抽象基类
WordQuery	Query_base 的派生类，用于查找一个给定的单词
NotQuery	Query_base 的派生类，查询结果是 Query 运算对象没有出现的行的集合
BinaryQuery	Query_base 派生出来的另一个抽象基类，表示有两个运算对象的查询
OrQuery	BinaryQuery 的派生类，返回它的两个运算对象分别出现的行的并集
AndQuery	BinaryQuery 的派生类，返回它的两个运算对象分别出现的行的交集

15.9.1 节练习

练习 15.31：已知 s1、s2、s3 和 s4 都是 string，判断下面的表达式分别创建了什么样的对象：

(a) `Query(s1) | Query(s2) & ~ Query(s3);`
(b) `Query(s1) | (Query(s2) & ~ Query(s3));`
(c) `(Query(s1) & (Query(s2)) | (Query(s3) & Query(s4)));`

15.9.2 Query_base 类和 Query 类

下面我们开始程序的实现过程，首先定义 Query_base 类：

```
// 这是一个抽象基类，具体的查询类型从中派生，所有成员都是 private 的
class Query_base {
    friend class Query;
protected:
    using line_no = TextQuery::line_no;   // 用于 eval 函数
    virtual ~Query_base() = default;
private:
    // eval 返回与当前 Query 匹配的 QueryResult
    virtual QueryResult eval(const TextQuery&) const = 0;
    // rep 是表示查询的一个 string
    virtual std::string rep() const = 0;
};
```

eval 和 rep 都是纯虚函数，因此 Query_base 是一个抽象基类（参见 15.4 节，第 541 页）。因为我们不希望用户或者派生类直接使用 Query_base，所以它没有 public 成员。所有对 Query_base 的使用都需要通过 Query 对象，因为 Query 需要调用 Query_base 的虚函数，所以我们将 Query 声明成 Query_base 的友元。

受保护的成员 line_no 将在 eval 函数内部使用。类似的，析构函数也是受保护的，因为它将（隐式地）在派生类析构函数中使用。

Query 类

Query 类对外提供接口，同时隐藏了 Query_base 的继承体系。每个 Query 对象都含有一个指向 Query_base 对象的 shared_ptr。因为 Query 是 Query_base 的唯一接口，所以 Query 必须定义自己的 eval 和 rep 版本。

接受一个 string 参数的 Query 构造函数将创建一个新的 WordQuery 对象，然后将它的 shared_prt 成员绑定到这个新创建的对象上。&、| 和 ~ 运算符分别创建 AndQuery、OrQuery 和 NotQuery 对象，这些运算符将返回一个绑定到新创建的对象上的 Query 对象。为了支持这些运算符，Query 还需要另外一个构造函数，它接受指向 Query_base 的 shared_ptr 并且存储给定的指针。我们将这个构造函数声明为私有的，原因是我们不希望一般的用户代码能随便定义 Query_base 对象。因为这个构造函数是私有的，所以我们需要将三个运算符声明为友元。

在形成了上述设计思路后，Query 类本身就比较简单了：

```
// 这是一个管理 Query_base 继承体系的接口类
class Query {
    // 这些运算符需要访问接受 shared_ptr 的构造函数，而该函数是私有的
    friend Query operator~(const Query &);
    friend Query operator|(const Query&, const Query&);
    friend Query operator&(const Query&, const Query&);
public:
    Query(const std::string&);          // 构建一个新的 WordQuery
    // 接口函数：调用对应的 Query_base 操作
    QueryResult eval(const TextQuery &t) const
                            { return q->eval(t); }
    std::string rep() const { return q->rep(); }
```

641

```
private:
    Query(std::shared_ptr<Query_base> query): q(query) { }
    std::shared_ptr<Query_base> q;
};
```

我们首先将创建 Query 对象的运算符声明为友元，之所以这么做是因为这些运算符需要访问那个私有构造函数。

在 Query 的公有接口部分，我们声明了接受 string 的构造函数，不过没有对其进行定义。因为这个构造函数将要创建一个 WordQuery 对象，所以我们应该首先定义 WordQuery 类，随后才能定义接受 string 的 Query 构造函数。

另外两个公有成员是 Query_base 的接口。其中，Query 操作使用它的 Query_base 指针来调用各自的 Query_base 虚函数。实际调用哪个函数版本将由 q 所指的对象类型决定，并且直到运行时才能最终确定下来。

Query 的输出运算符

输出运算符可以很好地解释我们的整个查询系统是如何工作的：

```
std::ostream &
operator<<(std::ostream &os, const Query &query)
{
    // Query::rep 通过它的 Query_base 指针对 rep()进行了虚调用
    return os << query.rep();
}
```

当我们打印一个 Query 时，输出运算符调用 Query 类的公有 rep 成员。运算符函数通过指针成员虚调用当前 Query 所指对象的 rep 成员。也就是说，当我们编写如下代码时：

```
Query andq = Query(sought1) & Query(sought2);
cout << andq << endl;
```

输出运算符将调用 andq 的 Query::rep，而 Query::rep 通过它的 Query_base 指针虚调用 Query_base 版本的 rep 函数。因为 andq 指向的是一个 AndQuery 对象，所以本次的函数调用将运行 AndQuery::rep。

15.9.2 节练习

练习 15.32：当一个 Query 类型的对象被拷贝、移动、赋值或销毁时，将分别发生什么？

练习 15.33：当一个 Query_base 类型的对象被拷贝、移动、赋值或销毁时，将分别发生什么？

642

15.9.3 派生类

对于 Query_base 的派生类来说，最有趣的部分是这些派生类如何表示一个真实的查询。其中 WordQuery 类最直接，它的任务就是保存要查找的单词。

其他类分别操作一个或两个运算对象。NotQuery 有一个运算对象，AndQuery 和 OrQuery 有两个。在这些类当中，运算对象可以是 Query_base 的任意一个派生类的对象：一个 NotQuery 对象可以被用在 WordQuery、AndQuery、OrQuery 或另一个 NotQuery 中。为了支持这种灵活性，运算对象必须以 Query_base 指针的形式存储，

这样我们就能把该指针绑定到任何我们需要的具体类上。

然而，实际上我们的类并不存储 Query_base 指针，而是直接使用一个 Query 对象。就像用户代码可以通过接口类得到简化一样，我们也可以使用接口类来简化我们自己的类。

至此我们已经清楚了所有类的设计思路，接下来依次实现它们。

WordQuery 类

一个 WordQuery 查找一个给定的 string，它是在给定的 TextQuery 对象上实际执行查询的唯一一个操作：

```
class WordQuery: public Query_base {
    friend class Query;                // Query 使用 WordQuery 构造函数
    WordQuery(const std::string &s): query_word(s) { }
    // 具体的类：WordQuery 将定义所有继承而来的纯虚函数
    QueryResult eval(const TextQuery &t) const
                    { return t.query(query_word); }
    std::string rep() const { return query_word; }
    std::string query_word;            // 要查找的单词
};
```

和 Query_base 一样，WordQuery 没有公有成员。同时，Query 必须作为 WordQuery 的友元，这样 Query 才能访问 WordQuery 的构造函数。

每个表示具体查询的类都必须定义继承而来的纯虚函数 eval 和 rep。我们在 WordQuery 类的内部定义这两个操作：eval 调用其 TextQuery 参数的 query 成员，由 query 成员在文件中实际进行查找；rep 返回这个 WordQuery 表示的 string（即 query_word）。

定义了 WordQuery 类之后，我们就能定义接受 string 的 Query 构造函数了：

```
inline
Query::Query(const std::string &s): q(new WordQuery(s)) { }
```

这个构造函数分配一个 WordQuery，然后令其指针成员指向新分配的对象。

NotQuery 类及 ~ 运算符

~ 运算符生成一个 NotQuery，其中保存着一个需要对其取反的 Query：

```
class NotQuery: public Query_base {
    friend Query operator~(const Query &);
    NotQuery(const Query &q): query(q) { }
    // 具体的类：NotQuery 将定义所有继承而来的纯虚函数
    std::string rep() const {return "~(" + query.rep() + ")";}
    QueryResult eval(const TextQuery&) const;
    Query query;
};
inline Query operator~(const Query &operand)
{
    return std::shared_ptr<Query_base>(new NotQuery(operand));
}
```

643

因为 NotQuery 的所有成员都是私有的，所以我们一开始就要把 ~ 运算符设定为友元。为

了 rep 一个 NotQuery，我们需要将~符号与基础的 Query 连接在一起。我们在输出的结果中加上适当的括号，这样读者就可以清楚地知道查询的优先级了。

值得注意的是，在 NotQuery 自己的 rep 成员中对 rep 的调用最终执行的是一个虚调用：query.rep()是对 Query 类 rep 成员的非虚调用，接着 Query::rep 将调用 q->rep()，这是一个通过 Query_base 指针进行的虚调用。

~运算符动态分配一个新的 NotQuery 对象，其 return 语句隐式地使用接受一个 shared_ptr<Query_base>的 Query 构造函数。也就是说，return 语句等价于：

```
// 分配一个新的 NotQuery 对象
// 将所得的 NotQuery 指针绑定到一个 shared_ptr<Query_base>
shared_ptr<Query_base> tmp(new NotQuery(expr));
return Query(tmp);          // 使用接受一个 shared_ptr 的 Query 构造函数
```

eval 成员比较复杂，因此我们将在类的外部实现它，15.9.4 节（第 573 页）将专门介绍如何定义 eval 函数。

BinaryQuery 类

BinaryQuery 类也是一个抽象基类，它保存操作两个运算对象的查询类型所需的数据：

```
class BinaryQuery: public Query_base {
protected:
    BinaryQuery(const Query &l, const Query &r, std::string s):
          lhs(l), rhs(r), opSym(s) { }
    // 抽象类：BinaryQuery 不定义 eval
    std::string rep() const { return "(" + lhs.rep() + " "
                                         + opSym + " "
                                         + rhs.rep() + ")"; }
    Query lhs, rhs;             // 左侧和右侧运算对象
    std::string opSym;          // 运算符的名字
};
```

644 BinaryQuery 中的数据是两个运算对象及相应的运算符符号，构造函数负责接受两个运算对象和一个运算符符号，然后将它们存储在对应的数据成员中。

要想 rep 一个 BinaryQuery，我们需要生成一个带括号的表达式。表达式的内容依次包括左侧运算对象、运算符以及右侧运算对象。就像我们显示 NotQuery 的方法一样，对 rep 的调用最终是对 lhs 和 rhs 所指 Query_base 对象的 rep 函数进行虚调用。

BinaryQuery 不定义 eval，而是继承了该纯虚函数。因此，BinaryQuery 也是一个抽象基类，我们不能创建 BinaryQuery 类型的对象。

AndQuery 类、OrQuery 类及相应的运算符

AndQuery 类和 OrQuery 类以及它们的运算符都非常相似：

```
class AndQuery: public BinaryQuery {
    friend Query operator&(const Query&, const Query&);
    AndQuery(const Query &left, const Query &right):
                        BinaryQuery(left, right, "&") { }
    // 具体的类：AndQuery 继承了 rep 并且定义了其他纯虚函数
    QueryResult eval(const TextQuery&) const;
```

```
};
inline Query operator&(const Query &lhs, const Query &rhs)
{
    return std::shared_ptr<Query_base>(new AndQuery(lhs, rhs));
}

class OrQuery: public BinaryQuery {
    friend Query operator|(const Query&, const Query&);
    OrQuery(const Query &left, const Query &right):
                BinaryQuery(left, right, "|") { }
    QueryResult eval(const TextQuery&) const;
};
inline Query operator|(const Query &lhs, const Query &rhs)
{
    return std::shared_ptr<Query_base>(new OrQuery(lhs, rhs));
}
```

这两个类将各自的运算符定义成友元，并且各自定义了一个构造函数通过运算符创建 BinaryQuery 基类部分。它们继承 BinaryQuery 的 rep 函数，但是覆盖了 eval 函数。

和~运算符一样，&和|运算符也返回一个绑定到新分配对象上的 shared_ptr。在这些运算符中，return 语句负责将 shared_ptr 转换成 Query。

15.9.3 节练习

645

练习 15.34：针对图 15.3（第 565 页）构建的表达式：

(a) 列举出在处理表达式的过程中执行的所有构造函数。
(b) 列举出 cout<<q 所调用的 rep。
(c) 列举出 q.eval()所调用的 eval。

练习 15.35：实现 Query 类和 Query_base 类，其中需要定义 rep 而无须定义 eval。

练习 15.36：在构造函数和 rep 成员中添加打印语句，运行你的代码以检验你对本节第一个练习中（a）、（b）两小题的回答是否正确。

练习 15.37：如果在派生类中含有 shared_ptr<Query_base>类型的成员而非 Query 类型的成员，则你的类需要做出怎样的改变？

练习 15.38：下面的声明合法吗？如果不合法，请解释原因；如果合法，请指出该声明的含义。

```
BinaryQuery a = Query("fiery") & Query("bird");
AndQuery b = Query("fiery") & Query("bird");
OrQuery c = Query("fiery") & Query("bird");
```

15.9.4 eval 函数

eval 函数是我们这个查询系统的核心。每个 eval 函数作用于各自的运算对象，同时遵循的内在逻辑也有所区别：OrQuery 的 eval 操作返回两个运算对象查询结果的并集，而 AndQuery 返回交集。与它们相比，NotQuery 的 eval 函数更加复杂一些：它需要返回运算对象没有出现的文本行。

为了支持上述 eval 函数的处理，我们需要使用 QueryResult，在它当中定义了 12.3.2 节练习（第 435 页）添加的成员。假设 QueryResult 包含 begin 和 end 成员，它们允许我们在 QueryResult 保存的行号 set 中进行迭代；另外假设 QueryResult 还包含一个名为 get_file 的成员，它返回一个指向待查询文件的 shared_ptr。

我们的 Query 类使用了 12.3.2 节练习（第 435 页）为 QueryResult 定义的成员。

OrQuery::eval

一个 OrQuery 表示的是它的两个运算对象结果的并集，对于每个运算对象来说，我们通过调用 eval 得到它的查询结果。因为这些运算对象的类型是 Query，所以调用 eval 也就是调用 Query::eval，而后者实际上是对潜在的 Query_base 对象的 eval 进行虚调用。每次调用完成后，得到的结果是一个 QueryResult，它表示运算对象出现的行号。我们把这些行号组织在一个新 set 中：

646

```
// 返回运算对象查询结果 set 的并集
QueryResult
OrQuery::eval(const TextQuery& text) const
{
    // 通过 Query 成员 lhs 和 rhs 进行的虚调用
    // 调用 eval 返回每个运算对象的 QueryResult
    auto right = rhs.eval(text), left = lhs.eval(text);
    // 将左侧运算对象的行号拷贝到结果 set 中
    auto ret_lines =
         make_shared<set<line_no>>(left.begin(), left.end());
    // 插入右侧运算对象所得的行号
    ret_lines->insert(right.begin(), right.end());
    // 返回一个新的 QueryResult，它表示 lhs 和 rhs 的并集
    return QueryResult(rep(), ret_lines, left.get_file());
}
```

我们使用接受一对迭代器的 set 构造函数初始化 ret_lines。一个 QueryResult 的 begin 和 end 成员返回行号 set 的迭代器，因此，创建 ret_lines 的过程实际上是拷贝了 left 集合的元素。接下来对 ret_lines 调用 insert，并将 right 的元素插入进来。调用结束后，ret_lines 将包含在 left 或 right 中出现过的所有行号。

eval 函数在最后构建并返回一个表示混合查询匹配的 QueryResult。QueryResult 的构造函数（参见 12.3.2 节，第 434 页）接受三个实参：一个表示查询的 string、一个指向匹配行号 set 的 shared_ptr 和一个指向输入文件 vector 的 shared_ptr。我们调用 rep 生成所需的 string，调用 get_file 获取指向文件的 shared_ptr。因为 left 和 right 指向的是同一个文件，所以使用哪个执行 get_file 函数并不重要。

AndQuery::eval

AndQuery 的 eval 和 OrQuery 很类似，唯一的区别是它调用了一个标准库算法来求得两个查询结果中共有的行：

```
// 返回运算对象查询结果 set 的交集
QueryResult
AndQuery::eval(const TextQuery& text) const
{
```

```
    // 通过 Query 运算对象进行的虚调用，以获得运算对象的查询结果 set
    auto left = lhs.eval(text), right = rhs.eval(text);
    // 保存 left 和 right 交集的 set
    auto ret_lines = make_shared<set<line_no>>();
    // 将两个范围的交集写入一个目的迭代器中
    // 本次调用的目的迭代器向 ret 添加元素
    set_intersection(left.begin(), left.end(),
                     right.begin(), right.end(),
                     inserter(*ret_lines, ret_lines->begin()));
    return QueryResult(rep(), ret_lines, left.get_file());
}
```

其中我们使用标准库算法 set_intersection 来合并两个 set，关于 set_intersection 在附录 A.2.8（第 779 页）中有详细的描述。 647

set_intersection 算法接受五个迭代器。它使用前四个迭代器表示两个输入序列（参见 10.5.2 节，第 368 页），最后一个实参表示目的位置。该算法将两个输入序列中共同出现的元素写入到目的位置中。

在上述调用中我们传入一个插入迭代器（参见 10.4.1 节，第 357 页）作为目的位置。当 set_intersection 向这个迭代器写入内容时，实际上是向 ret_lines 插入一个新元素。

和 OrQuery 的 eval 函数一样，AndQuery 的 eval 函数也在最后构建并返回一个表示混合查询匹配的 QueryResult。

NotQuery::eval

NotQuery 查找运算对象没有出现的文本行：

```
// 返回运算对象的结果 set 中不存在的行
QueryResult
NotQuery::eval(const TextQuery& text) const
{
    // 通过 Query 运算对象对 eval 进行虚调用
    auto result = query.eval(text);
    // 开始时结果 set 为空
    auto ret_lines = make_shared<set<line_no>>();
    // 我们必须在运算对象出现的所有行中进行迭代
    auto beg = result.begin(), end = result.end();
    // 对于输入文件的每一行，如果该行不在 result 当中，则将其添加到 ret_lines
    auto sz = result.get_file()->size();
    for (size_t n = 0; n != sz; ++n) {
        // 如果我们还没有处理完 result 的所有行
        // 检查当前行是否存在
        if (beg == end || *beg != n)
            ret_lines->insert(n);     // 如果不在 result 当中，添加这一行
        else if (beg != end)
            ++beg;        // 否则继续获取 result 的下一行（如果有的话）
    }
    return QueryResult(rep(), ret_lines, result.get_file());
}
```

和其他 eval 函数一样，我们首先对当前的运算对象调用 eval，所得的结果

QueryResult 中包含的是运算对象出现的行号，但我们想要的是运算对象未出现的行号。也就是说，我们需要的是存在于文件中，但是不在 result 中的行。

要想得到最终的结果，我们需要遍历不超过输出文件大小的所有整数，并将所有不在 result 中的行号放入到 ret_lines 中。我们使用 beg 和 end 分别表示 result 的第一个元素和最后一个元素的下一位置。因为遍历的对象是一个 set，所以当遍历结束后获得的行号将按照升序排列。

648

循环体负责检查当前的编号是否在 result 当中。如果不在，将这个数字添加到 ret_lines 中；如果该数字属于 result，则我们递增 result 的迭代器 beg。

一旦处理完所有行号，就返回包含 ret_lines 的一个 QueryResult 对象；和之前版本的 eval 类似，该 QueryResult 对象还包含 rep 和 get_file 的运行结果。

15.9.4 节练习

练习 15.39：实现 Query 类和 Query_base 类，求图 15.3（第 565 页）中表达式的值并打印相关信息，验证你的程序是否正确。

练习 15.40：在 OrQuery 的 eval 函数中，如果 rhs 成员返回的是空集将发生什么？如果 lhs 是空集呢？如果 lhs 和 rhs 都是空集又将发生什么？

练习 15.41：重新实现你的类，这次使用指向 Query_base 的内置指针而非 shared_ptr。请注意，做出上述改动后你的类将不能再使用合成的拷贝控制成员。

练习 15.42：从下面的几种改进中选择一种，设计并实现它：

(a) 按句子查询并打印单词，而不再是按行打印。

(b) 引入一个历史系统，用户可以按编号查阅之前的某个查询，并可以在其中增加内容或者将其与其他查询组合。

(c) 允许用户对结果做出限制，比如从给定范围的行中挑出匹配的进行显示。

小结

649

继承使得我们可以编写一些新的类，这些新类既能共享其基类的行为，又能根据需要覆盖或添加行为。动态绑定使得我们可以忽略类型之间的差异，其机理是在运行时根据对象的动态类型来选择运行函数的哪个版本。继承和动态绑定的结合使得我们能够编写具有特定类型行为但又独立于类型的程序。

在C++语言中，动态绑定只作用于虚函数，并且需要通过指针或引用调用。

在派生类对象中包含有与它的每个基类对应的子对象。因为所有派生类对象都含有基类部分，所以我们能将派生类的引用或指针转换为一个可访问的基类引用或指针。

当执行派生类的构造、拷贝、移动和赋值操作时，首先构造、拷贝、移动和赋值其中的基类部分，然后才轮到派生类部分。析构函数的执行顺序则正好相反，首先销毁派生类，接下来执行基类子对象的析构函数。基类通常都应该定义一个虚析构函数，即使基类根本不需要析构函数也最好这么做。将基类的析构函数定义成虚函数的原因是为了确保当我们删除一个基类指针，而该指针实际指向一个派生类对象时，程序也能正确运行。

派生类为它的每个基类提供一个保护级别。`public`基类的成员也是派生类接口的一部分；`private`基类的成员是不可访问的；`protected`基类的成员对于派生类的派生类是可访问的，但是对于派生类的用户不可访问。

术语表

抽象基类（abstract base class） 含有一个或多个纯虚函数的类，我们无法创建抽象基类的对象。

可访问的（accessible） 能被派生类对象访问的基类成员。可访问性由派生类的派生列表中所用的访问说明符和基类中成员的访问级别共同决定。例如，通过公有继承而来的一个公有成员对于派生类的用户来说是可访问的；而私有继承而来的公有成员是不可访问的。

基类（base class） 可供其他类继承的类。基类的成员也将成为派生类的成员。

类派生列表（class derivation list） 罗列了所有基类，每个基类包含一个可选的访问级别，它定义了派生类继承该基类的方式。如果没有提供访问说明符，则当派生类通过关键字`struct`定义时继承是公有的；而当派生类通过关键字`class`定义时继承是私有的。

派生类（derived class） 从其他类派生而来的类。派生类可以覆盖其基类的虚函数，也可以定义自己的新成员。派生类的作用域嵌套在基类作用域当中；派生类的成员能直接访问基类的成员。

派生类向基类的类型转换（derived-to-base conversion） 派生类对象向基类引用或者派生类指针向基类指针的隐式类型转换。

直接基类（direct base class） 派生类直接继承的基类，直接基类在派生类的派生列表中说明。直接基类本身也可以是一个派生类。

650

动态绑定（dynamic binding） 直到运行时才确定到底执行函数的哪个版本。在C++语言中，动态绑定的意思是在运行时根据引用或指针所绑定对象的实际类型来选择执行虚函数的某一个版本。

动态类型（dynamic type） 对象在运行时的类型。引用所引对象或者指针所指对象的动态类型可能与该引用或指针的静态类型不同。基类的指针或引用可以指向一个

派生类对象。在这样的情况中，静态类型是基类的引用（或指针），而动态类型是派生类的引用（或指针）。

间接基类（indirect base class） 不出现在派生类的派生列表中的基类。直接基类以直接或间接方式继承的类是派生类的间接基类。

继承（inheritance）由一个已有的类（基类）定义一个新类（派生类）的编程技术。派生类将继承基类的成员。

面向对象编程（object-oriented programming） 利用数据抽象、继承以及动态绑定等技术编写程序的方法。

覆盖（override） 派生类中定义的虚函数如果与基类中定义的同名虚函数有相同的形参列表，则派生类版本将覆盖基类的版本。

多态性（polymorphism） 当用于面向对象编程的范畴时，多态性的含义是指程序能通过引用或指针的动态类型获取类型特定行为的能力。

私有继承（private inheritance） 在私有继承中，基类的公有成员和受保护成员是派生类的私有成员。

protected 访问说明符（protected access specifier） `protected` 关键字之后定义的成员能被派生类的成员和友元访问。但是这些成员只对派生类对象是可访问的，对类的普通用户则是不可访问的。

受保护的继承（protected inheritance） 在受保护的继承中，基类的公有成员和受保护成员是派生类的受保护成员。

公有继承（public inheritance） 基类的公有接口是派生类公有接口的组成部分。

纯虚函数（pure virtual） 在类的内部声明虚函数时，在分号之前使用了`=0`。一个纯虚函数不需要（但是可以）被定义。含有纯虚函数的类是抽象基类。如果派生类没有对继承而来的纯虚函数定义自己的版本，则该派生类也是抽象的。

重构（refactoring） 重新设计程序以便将一些相关的部分搜集到一个单独的抽象中，然后使用新的抽象替换原来的代码。通常情况下，重构类的方式是将数据成员和函数成员移动到继承体系的高级别节点当中，从而避免代码冗余。

运行时绑定（run-time binding） 参见“动态绑定”。

切掉（sliced down） 当我们用一个派生类对象初始化基类对象或者为基类对象赋值时发生的情况。对象的派生类部分将被“切掉”，只剩下基类部分赋值给基类对象。

静态类型（static type） 对象被定义的类型或表达式产生的类型。静态类型在编译时是已知的。

虚函数（virtual function） 用于定义类型特定行为的成员函数。通过引用或指针对虚函数的调用直到运行时才被解析，依据是引用或指针所绑定对象的类型。

第 16 章
模板与泛型编程

内容

16.1 定义模板 578
16.2 模板实参推断 600
16.3 重载与模板 614
16.4 可变参数模板 618
16.5 模板特例化 624
小结 630
术语表 630

面向对象编程（OOP）和泛型编程都能处理在编写程序时不知道类型的情况。不同之处在于：OOP 能处理类型在程序运行之前都未知的情况；而在泛型编程中，在编译时就能获知类型了。

本书第 II 部分中介绍的容器、迭代器和算法都是泛型编程的例子。当我们编写一个泛型程序时，是独立于任何特定类型来编写代码的。当使用一个泛型程序时，我们提供类型或值，程序实例可在其上运行。

例如，标准库为每个容器提供了单一的、泛型的定义，如 `vector`。我们可以使用这个泛型定义来定义很多类型的 `vector`，它们的差异就在于包含的元素类型不同。

模板是泛型编程的基础。我们不必了解模板是如何定义的就能使用它们，实际上我们已经这样用了。在本章中，我们将学习如何定义自己的模板。

652 模板是 C++中泛型编程的基础。一个模板就是一个创建类或函数的蓝图或者说公式。当使用一个 vector 这样的泛型类型，或者 find 这样的泛型函数时，我们提供足够的信息，将蓝图转换为特定的类或函数。这种转换发生在编译时。在本书第 3 章和第 II 部分中我们已经学习了如何使用模板。在本章中，我们将学习如何定义模板。

16.1 定义模板

假定我们希望编写一个函数来比较两个值，并指出第一个值是小于、等于还是大于第二个值。在实际中，我们可能想要定义多个函数，每个函数比较一种给定类型的值。我们的初次尝试可能定义多个重载函数：

```
// 如果两个值相等，返回 0，如果 v1 小返回-1，如果 v2 小返回 1
int compare(const string &v1, const string &v2)
{
    if (v1 < v2) return -1;
    if (v2 < v1) return 1;
    return 0;
}
int compare(const double &v1, const double &v2)
{
    if (v1 < v2) return -1;
    if (v2 < v1) return 1;
    return 0;
}
```

这两个函数几乎是相同的，唯一的差异是参数的类型，函数体则完全一样。

如果对每种希望比较的类型都不得不重复定义完全一样的函数体，是非常烦琐且容易出错的。更麻烦的是，在编写程序的时候，我们就要确定可能要 compare 的所有类型。如果希望能在用户提供的类型上使用此函数，这种策略就失效了。

16.1.1 函数模板

我们可以定义一个通用的**函数模板**（function template），而不是为每个类型都定义一个新函数。一个函数模板就是一个公式，可用来生成针对特定类型的函数版本。compare 的模板版本可能像下面这样：

```
template <typename T>
int compare(const T &v1, const T &v2)
{
    if (v1 < v2) return -1;
    if (v2 < v1) return 1;
    return 0;
}
```

653 模板定义以关键字 template 开始，后跟一个**模板参数列表**（template parameter list），这是一个逗号分隔的一个或多个**模板参数**（template parameter）的列表，用小于号（<）和大于号（>）包围起来。

在模板定义中，模板参数列表不能为空。

模板参数列表的作用很像函数参数列表。函数参数列表定义了若干特定类型的局部变量，但并未指出如何初始化它们。在运行时，调用者提供实参来初始化形参。

类似的，模板参数表示在类或函数定义中用到的类型或值。当使用模板时，我们（隐式地或显式地）指定**模板实参**（template argument），将其绑定到模板参数上。

我们的 compare 函数声明了一个名为 T 的类型参数。在 compare 中，我们用名字 T 表示一个类型。而 T 表示的实际类型则在编译时根据 compare 的使用情况来确定。

实例化函数模板

当我们调用一个函数模板时，编译器（通常）用函数实参来为我们推断模板实参。即，当我们调用 compare 时，编译器使用实参的类型来确定绑定到模板参数 T 的类型。例如，在下面的调用中：

```
cout << compare(1, 0) << endl; // T 为 int
```

实参类型是 int。编译器会推断出模板实参为 int，并将它绑定到模板参数 T。

编译器用推断出的模板参数来为我们**实例化**（instantiate）一个特定版本的函数。当编译器实例化一个模板时，它使用实际的模板实参代替对应的模板参数来创建出模板的一个新“实例”。例如，给定下面的调用：

```
// 实例化出 int compare(const int&, const int&)
cout << compare(1, 0) << endl; // T 为 int
// 实例化出 int compare(const vector<int>&, const vector<int>&)
vector<int> vec1{1, 2, 3}, vec2{4, 5, 6};
cout << compare(vec1, vec2) << endl; // T 为 vector<int>
```

编译器会实例化出两个不同版本的 compare。对于第一个调用，编译器会编写并编译一个 compare 版本，其中 T 被替换为 int：

```
int compare(const int &v1, const int &v2)
{
    if (v1 < v2) return -1;
    if (v2 < v1) return 1;
    return 0;
}
```

对于第二个调用，编译器会生成另一个 compare 版本，其中 T 被替换为 vector<int>。这些编译器生成的版本通常被称为模板的**实例**（instantiation）。

模板类型参数

654

我们的 compare 函数有一个模板**类型参数**（type parameter）。一般来说，我们可以将类型参数看作类型说明符，就像内置类型或类类型说明符一样使用。特别是，类型参数可以用来指定返回类型或函数的参数类型，以及在函数体内用于变量声明或类型转换：

```
// 正确：返回类型和参数类型相同
template <typename T> T foo(T* p)
{
    T tmp = *p; // tmp 的类型将是指针 p 指向的类型
    //...
    return tmp;
}
```

类型参数前必须使用关键字 class 或 typename：

```
// 错误：U 之前必须加上 class 或 typename
template <typename T, U> T calc(const T&, const U&);
```

在模板参数列表中，这两个关键字的含义相同，可以互换使用。一个模板参数列表中可以同时使用这两个关键字：

```
// 正确：在模板参数列表中，typename 和 class 没有什么不同
template <typename T, class U> calc (const T&, const U&);
```

看起来用关键字 typename 来指定模板类型参数比用 class 更为直观。毕竟，我们可以用内置（非类）类型作为模板类型实参。而且，typename 更清楚地指出随后的名字是一个类型名。但是，typename 是在模板已经广泛使用之后才引入 C++语言的，某些程序员仍然只用 class。

非类型模板参数

除了定义类型参数，还可以在模板中定义**非类型参数**（nontype parameter）。一个非类型参数表示一个值而非一个类型。我们通过一个特定的类型名而非关键字 class 或 typename 来指定非类型参数。

当一个模板被实例化时，非类型参数被一个用户提供的或编译器推断出的值所代替。这些值必须是常量表达式（参见 2.4.4 节，第 58 页），从而允许编译器在编译时实例化模板。

例如，我们可以编写一个 compare 版本处理字符串字面常量。这种字面常量是 const char 的数组。由于不能拷贝一个数组，所以我们将自己的参数定义为数组的引用（参见 6.2.4 节，第 195 页）。由于我们希望能比较不同长度的字符串字面常量，因此为模板定义了两个非类型的参数。第一个模板参数表示第一个数组的长度，第二个参数表示第二个数组的长度：

655

```
template<unsigned N, unsigned M>
int compare(const char (&p1)[N], const char (&p2)[M])
{
    return strcmp(p1, p2);
}
```

当我们调用这个版本的 compare 时：

```
compare("hi", "mom")
```

编译器会使用字面常量的大小来代替 N 和 M，从而实例化模板。记住，编译器会在一个字符串字面常量的末尾插入一个空字符作为终结符（参见 2.1.3 节，第 36 页），因此编译器会实例化出如下版本：

```
int compare(const char (&p1)[3], const char (&p2)[4])
```

一个非类型参数可以是一个整型，或者是一个指向对象或函数类型的指针或（左值）引用。绑定到非类型整型参数的实参必须是一个常量表达式。绑定到指针或引用非类型参数的实参必须具有静态的生存期（参见第 12 章，第 400 页）。我们不能用一个普通（非 static）局部变量或动态对象作为指针或引用非类型模板参数的实参。指针参数也可以用 nullptr 或一个值为 0 的常量表达式来实例化。

在模板定义内，模板非类型参数是一个常量值。在需要常量表达式的地方，可以使用

非类型参数，例如，指定数组大小。

非类型模板参数的模板实参必须是常量表达式。

inline 和 constexpr 的函数模板

函数模板可以声明为 inline 或 constexpr 的，如同非模板函数一样。inline 或 constexpr 说明符放在模板参数列表之后，返回类型之前：

```
// 正确：inline 说明符跟在模板参数列表之后
template <typename T> inline T min(const T&, const T&);
// 错误：inline 说明符的位置不正确
inline template <typename T> T min(const T&, const T&);
```

编写类型无关的代码

我们最初的 compare 函数虽然简单，但它说明了编写泛型代码的两个重要原则：

- 模板中的函数参数是 const 的引用。
- 函数体中的条件判断仅使用<比较运算。

通过将函数参数设定为 const 的引用，我们保证了函数可以用于不能拷贝的类型。大多数类型，包括内置类型和我们已经用过的标准库类型（除 unique_ptr 和 IO 类型之外），都是允许拷贝的。但是，不允许拷贝的类类型也是存在的。通过将参数设定为 const 的引用，保证了这些类型可以用我们的 compare 函数来处理。而且，如果 compare 用于处理大对象，这种设计策略还能使函数运行得更快。 656

你可能认为既使用<运算符又使用>运算符来进行比较操作会更为自然：

```
// 期望的比较操作
if (v1 < v2) return -1;
if (v1 > v2) return 1;
return 0;
```

但是，如果编写代码时只使用<运算符，我们就降低了 compare 函数对要处理的类型的要求。这些类型必须支持<，但不必同时支持>。

实际上，如果我们真的关心类型无关和可移植性，可能需要用 less（参见 14.8.2 节，第 510 页）来定义我们的函数：

```
// 即使用于指针也正确的 compare 版本；参见 14.8.2 节（第 510 页）
template <typename T> int compare(const T &v1, const T &v2)
{
    if (less<T>()(v1, v2)) return -1;
    if (less<T>()(v2, v1)) return 1;
    return 0;
}
```

原始版本存在的问题是，如果用户调用它比较两个指针，且两个指针未指向相同的数组，则代码的行为是未定义的（据查阅资料，less<T>的默认实现用的就是<，所以这其实并未起到让这种比较有一个良好定义的作用——译者注）。

模板程序应该尽量减少对实参类型的要求。

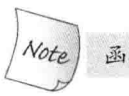
模板编译

当编译器遇到一个模板定义时，它并不生成代码。只有当我们实例化出模板的一个特定版本时，编译器才会生成代码。当我们使用（而不是定义）模板时，编译器才生成代码，这一特性影响了我们如何组织代码以及错误何时被检测到。

通常，当我们调用一个函数时，编译器只需要掌握函数的声明。类似的，当我们使用一个类类型的对象时，类定义必须是可用的，但成员函数的定义不必已经出现。因此，我们将类定义和函数声明放在头文件中，而普通函数和类的成员函数的定义放在源文件中。

模板则不同：为了生成一个实例化版本，编译器需要掌握函数模板或类模板成员函数的定义。因此，与非模板代码不同，模板的头文件通常既包括声明也包括定义。

657



函数模板和类模板成员函数的定义通常放在头文件中。

关键概念：模板和头文件

模板包含两种名字：

- 那些不依赖于模板参数的名字
- 那些依赖于模板参数的名字

当使用模板时，所有不依赖于模板参数的名字都必须是可见的，这是由模板的提供者来保证的。而且，模板的提供者必须保证，当模板被实例化时，模板的定义，包括类模板的成员的定义，也必须是可见的。

用来实例化模板的所有函数、类型以及与类型关联的运算符的声明都必须是可见的，这是由模板的用户来保证的。

通过组织良好的程序结构，恰当使用头文件，这些要求都很容易满足。模板的设计者应该提供一个头文件，包含模板定义以及在类模板或成员定义中用到的所有名字的声明。模板的用户必须包含模板的头文件，以及用来实例化模板的任何类型的头文件。

大多数编译错误在实例化期间报告

模板直到实例化时才会生成代码，这一特性影响了我们何时才会获知模板内代码的编译错误。通常，编译器会在三个阶段报告错误。

第一个阶段是编译模板本身时。在这个阶段，编译器通常不会发现很多错误。编译器可以检查语法错误，例如忘记分号或者变量名拼错等，但也就这么多了。

第二个阶段是编译器遇到模板使用时。在此阶段，编译器仍然没有很多可检查的。对于函数模板调用，编译器通常会检查实参数目是否正确。它还能检查参数类型是否匹配。对于类模板，编译器可以检查用户是否提供了正确数目的模板实参，但也仅限于此了。

第三个阶段是模板实例化时，只有这个阶段才能发现类型相关的错误。依赖于编译器如何管理实例化，这类错误可能在链接时才报告。

当我们编写模板时，代码不能是针对特定类型的，但模板代码通常对其所使用的类型有一些假设。例如，我们最初的 compare 函数中的代码就假定实参类型定义了<运算符。

```
if (v1 < v2) return -1;   // 要求类型 T 的对象支持<操作
if (v2 < v1) return 1;    // 要求类型 T 的对象支持<操作
```

```
    return 0;                // 返回 int；不依赖于 T
```

当编译器处理此模板时，它不能验证 if 语句中的条件是否合法。如果传递给 compare 的实参定义了<运算符，则代码就是正确的，否则就是错误的。例如，

658

```
    Sales_data data1, data2;
    cout << compare(data1, data2) << endl; // 错误：Sales_data 未定义<
```

此调用实例化了 compare 的一个版本，将 T 替换为 Sales_data。if 条件试图对 Sales_data 对象使用<运算符，但 Sales_data 并未定义此运算符。此实例化生成了一个无法编译通过的函数版本。但是，这样的错误直至编译器在类型 Sales_data 上实例化 compare 时才会被发现。

保证传递给模板的实参支持模板所要求的操作，以及这些操作在模板中能正确工作，是调用者的责任。

16.1.1 节练习

练习 16.1：给出实例化的定义。

练习 16.2：编写并测试你自己版本的 compare 函数。

练习 16.3：对两个 Sales_data 对象调用你的 compare 函数，观察编译器在实例化过程中如何处理错误。

练习 16.4：编写行为类似标准库 find 算法的模板。函数需要两个模板类型参数，一个表示函数的迭代器参数，另一个表示值的类型。使用你的函数在一个 vector<int>和一个 list<string>中查找给定值。

练习 16.5：为 6.2.4 节（第 195 页）中的 print 函数编写模板版本，它接受一个数组的引用，能处理任意大小、任意元素类型的数组。

练习 16.6：你认为接受一个数组实参的标准库函数 begin 和 end 是如何工作的？定义你自己版本的 begin 和 end。

练习 16.7：编写一个 constexpr 模板，返回给定数组的大小。

练习 16.8：在第 97 页的“关键概念”中，我们注意到，C++程序员喜欢使用!=而不喜欢<。解释这个习惯的原因。

16.1.2 类模板

类模板（class template）是用来生成类的蓝图的。与函数模板的不同之处是，编译器不能为类模板推断模板参数类型。如我们已经多次看到的，为了使用类模板，我们必须在模板名后的尖括号中提供额外信息（参见 3.3 节，第 87 页）——用来代替模板参数的模板实参列表。

659

定义类模板

作为一个例子，我们将实现 StrBlob（参见 12.1.1 节，第 405 页）的模板版本。我们将此模板命名为 Blob，意指它不再针对 string。类似 StrBlob，我们的模板会提供对元素的共享（且核查过的）访问能力。与类不同，我们的模板可以用于更多类型的元素。与标准库容器相同，当使用 Blob 时，用户需要指出元素类型。

类似函数模板，类模板以关键字 template 开始，后跟模板参数列表。在类模板（及其成员）的定义中，我们将模板参数当作替身，代替使用模板时用户需要提供的类型或值：

```
template <typename T> class Blob {
public:
    typedef T value_type;
    typedef typename std::vector<T>::size_type size_type;
    // 构造函数
    Blob();
    Blob(std::initializer_list<T> il);
    // Blob 中的元素数目
    size_type size() const { return data->size(); }
    bool empty() const { return data->empty(); }
    // 添加和删除元素
    void push_back(const T &t) {data->push_back(t);}
    // 移动版本，参见 13.6.3 节（第 484 页）
    void push_back(T &&t) { data->push_back(std::move(t)); }
    void pop_back();
    // 元素访问
    T& back();
    T& operator[](size_type i); // 在 14.5 节(第 501 页)中定义
private:
    std::shared_ptr<std::vector<T>> data;
    // 若 data[i]无效，则抛出 msg
    void check(size_type i, const std::string &msg) const;
};
```

我们的 Blob 模板有一个名为 T 的模板类型参数，用来表示 Blob 保存的元素的类型。例如，我们将元素访问操作的返回类型定义为 T&。当用户实例化 Blob 时，T 就会被替换为特定的模板实参类型。

除了模板参数列表和使用 T 代替 string 之外，此类模板的定义与 12.1.1 节（第 405 页）中定义的类版本及 12.1.6 节（第 422 页）和第 13 章、第 14 章中更新的版本是一样的。

660

实例化类模板

我们已经多次见到，当使用一个类模板时，我们必须提供额外信息。我们现在知道这些额外信息是**显式模板实参**（explicit template argument）列表，它们被绑定到模板参数。编译器使用这些模板实参来实例化出特定的类。

例如，为了用我们的 Blob 模板定义一个类型，必须提供元素类型：

```
Blob<int> ia;                  // 空 Blob<int>
Blob<int> ia2 = {0,1,2,3,4}; // 有 5 个元素的 Blob<int>
```

ia 和 ia2 使用相同的特定类型版本的 Blob（即 Blob<int>）。从这两个定义，编译器会实例化出一个与下面定义等价的类：

```
template <> class Blob<int> {
    typedef typename std::vector<int>::size_type size_type;
    Blob();
    Blob(std::initializer_list<int> il);
    //...
    int& operator[](size_type i);
```

```
private:
    std::shared_ptr<std::vector<int>> data;
    void check(size_type i, const std::string &msg) const;
};
```

当编译器从我们的 Blob 模板实例化出一个类时，它会重写 Blob 模板，将模板参数 T 的每个实例替换为给定的模板实参，在本例中是 int。

对我们指定的每一种元素类型，编译器都生成一个不同的类：

```
// 下面的定义实例化出两个不同的 Blob 类型
Blob<string> names; // 保存 string 的 Blob
Blob<double> prices;// 不同的元素类型
```

这两个定义会实例化出两个不同的类。names 的定义创建了一个 Blob 类，每个 T 都被替换为 string。prices 的定义生成了另一个 Blob 类，T 被替换为 double。

一个类模板的每个实例都形成一个独立的类。类型 Blob<string>与任何其他 Blob 类型都没有关联，也不会对任何其他 Blob 类型的成员有特殊访问权限。

在模板作用域中引用模板类型

为了阅读模板类代码，应该记住类模板的名字不是一个类型名（参见 3.3 节，第 87 页）。类模板用来实例化类型，而一个实例化的类型总是包含模板参数的。

可能令人迷惑的是，一个类模板中的代码如果使用了另外一个模板，通常不将一个实际类型（或值）的名字用作其模板实参。相反的，我们通常将模板自己的参数当作被使用模板的实参。例如，我们的 data 成员使用了两个模板，vector 和 shared_ptr。我们知道，无论何时使用模板都必须提供模板实参。在本例中，我们提供的模板实参就是 Blob 的模板参数。因此，data 的定义如下：

661

```
std::shared_ptr<std::vector<T>> data;
```

它使用了 Blob 的类型参数来声明 data 是一个 shared_ptr 的实例，此 shared_ptr 指向一个保存类型为 T 的对象的 vector 实例。当我们实例化一个特定类型的 Blob，例如 Blob<string>时，data 会成为：

```
shared_ptr<vector<string>>
```

如果我们实例化 Blob<int>，则 data 会成为 shared_ptr<vector<int>>，依此类推。

类模板的成员函数

与其他任何类相同，我们既可以在类模板内部，也可以在类模板外部为其定义成员函数，且定义在类模板内的成员函数被隐式声明为内联函数。

类模板的成员函数本身是一个普通函数。但是，类模板的每个实例都有其自己版本的成员函数。因此，类模板的成员函数具有和模板相同的模板参数。因而，定义在类模板之外的成员函数就必须以关键字 template 开始，后接类模板参数列表。

与往常一样，当我们在类外定义一个成员时，必须说明成员属于哪个类。而且，从一个模板生成的类的名字中必须包含其模板实参。当我们定义一个成员函数时，模板实参与模板形参相同。即，对于 StrBlob 的一个给定的成员函数

```
ret-type StrBlob::member-name(parm-list)
```

对应的 Blob 的成员应该是这样的：

```
template <typename T>
ret-type Blob<T>::member-name(parm-list)
```

check 和元素访问成员

我们首先定义 check 成员，它检查一个给定的索引：

```
template <typename T>
void Blob<T>::check(size_type i, const std::string &msg) const
{
    if (i >= data->size())
        throw std::out_of_range(msg);
}
```

除了类名中的不同之处以及使用了模板参数列表外，此函数与原 StrBlob 类的 check 成员完全一样。

下标运算符和 back 函数用模板参数指出返回类型，其他未变：

662

```
template <typename T>
T& Blob<T>::back()
{
    check(0, "back on empty Blob");
    return data->back();
}
template <typename T>
T& Blob<T>::operator[](size_type i)
{
    // 如果 i 太大，check 会抛出异常，阻止访问一个不存在的元素
    check(i, "subscript out of range");
    return (*data)[i];
}
```

在原 StrBlob 类中，这些运算符返回 string&。而模板版本则返回一个引用，指向用来实例化 Blob 的类型。

pop_back 函数与原 StrBlob 的成员几乎相同：

```
template <typename T> void Blob<T>::pop_back()
{
    check(0, "pop_back on empty Blob");
    data->pop_back();
}
```

在原 StrBlob 类中，下标运算符和 back 成员都对 const 对象进行了重载。我们将这些成员及 front 成员的定义留作练习。

Blob 构造函数

与其他任何定义在类模板外的成员一样，构造函数的定义要以模板参数开始：

```
template <typename T>
Blob<T>::Blob(): data(std::make_shared<std::vector<T>>()) { }
```

这段代码在作用域 Blob<T>中定义了名为 Blob 的成员函数。类似 StrBlob 的默认构造

函数（参见 12.1.1 节，第 405 页），此构造函数分配一个空 vector，并将指向 vector 的指针保存在 data 中。如前所述，我们将类模板自己的类型参数作为 vector 的模板实参来分配 vector。

类似的，接受一个 initializer_list 参数的构造函数将其类型参数 T 作为 initializer_list 参数的元素类型：

```
template <typename T>
Blob<T>::Blob(std::initializer_list<T> il):
              data(std::make_shared<std::vector<T>>(il)) { }
```

类似默认构造函数，此构造函数分配一个新的 vector。在本例中，我们用参数 il 来初始化此 vector。

为了使用这个构造函数，我们必须传递给它一个 initializer_list，其中的元素必须与 Blob 的元素类型兼容：

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

这条语句中，构造函数的参数类型为 initializer_list<string>。列表中的每个字符串字面常量隐式地转换为一个 string。

类模板成员函数的实例化

663

默认情况下，一个类模板的成员函数只有当程序用到它时才进行实例化。例如，下面代码

```
// 实例化 Blob<int>和接受 initializer_list<int>的构造函数
Blob<int> squares = {0,1,2,3,4,5,6,7,8,9};
// 实例化 Blob<int>::size() const
for (size_t i = 0; i != squares.size(); ++i)
    squares[i] = i*i; // 实例化 Blob<int>::operator[](size_t)
```

实例化了 Blob<int>类和它的三个成员函数：operator[]、size 和接受 initializer_list<int>的构造函数。

如果一个成员函数没有被使用，则它不会被实例化。成员函数只有在被用到时才进行实例化，这一特性使得即使某种类型不能完全符合模板操作的要求（参见 9.2 节，第 294 页），我们仍然能用该类型实例化类。

默认情况下，对于一个实例化了的类模板，其成员只有在使用时才被实例化。

在类代码内简化模板类名的使用

当我们使用一个类模板类型时必须提供模板实参，但这一规则有一个例外。在类模板自己的作用域中，我们可以直接使用模板名而不提供实参：

```
// 若试图访问一个不存在的元素，BlobPtr 抛出一个异常
template <typename T> class BlobPtr {
public:
    BlobPtr(): curr(0) { }
    BlobPtr(Blob<T> &a, size_t sz = 0):
            wptr(a.data), curr(sz) { }
    T& operator*() const
    { auto p = check(curr, "dereference past end");
      return (*p)[curr]; // (*p)为本对象指向的 vector
```

```
    }
    // 递增和递减
    BlobPtr& operator++(); // 前置运算符
    BlobPtr& operator--();
private:
    // 若检查成功，check 返回一个指向 vector 的 shared_ptr
    std::shared_ptr<std::vector<T>>
        check(std::size_t, const std::string&) const;
    // 保存一个 weak_ptr，表示底层 vector 可能被销毁
    std::weak_ptr<std::vector<T>> wptr;
    std::size_t curr; // 数组中的当前位置
};
```

细心的读者可能已经注意到，BlobPtr 的前置递增和递减成员返回 BlobPtr&，而不是 BlobPtr<T>&。当我们处于一个类模板的作用域中时，编译器处理模板自身引用时就好像我们已经提供了与模板参数匹配的实参一样。即，就好像我们这样编写代码一样：

664

```
BlobPtr<T>& operator++();
BlobPtr<T>& operator--();
```

在类模板外使用类模板名

当我们在类模板外定义其成员时，必须记住，我们并不在类的作用域中，直到遇到类名才表示进入类的作用域（参见 7.4 节，第 253 页）：

```
// 后置：递增/递减对象但返回原值
template <typename T>
BlobPtr<T> BlobPtr<T>::operator++(int)
{
    // 此处无须检查；调用前置递增时会进行检查
    BlobPtr ret = *this; // 保存当前值
    ++*this;     // 推进一个元素；前置++检查递增是否合法
    return ret;  // 返回保存的状态
}
```

由于返回类型位于类的作用域之外，我们必须指出返回类型是一个实例化的 BlobPtr，它所用类型与类实例化所用类型一致。在函数体内，我们已经进入类的作用域，因此在定义 ret 时无须重复模板实参。如果不提供模板实参，则编译器将假定我们使用的类型与成员实例化所用类型一致。因此，ret 的定义与如下代码等价：

```
BlobPtr<T> ret = *this;
```

在一个类模板的作用域内，我们可以直接使用模板名而不必指定模板实参。

类模板和友元

当一个类包含一个友元声明（参见 7.2.1 节，第 241 页）时，类与友元各自是否是模板是相互无关的。如果一个类模板包含一个非模板友元，则友元被授权可以访问所有模板实例。如果友元自身是模板，类可以授权给所有友元模板实例，也可以只授权给特定实例。

一对一友好关系

类模板与另一个（类或函数）模板间友好关系的最常见的形式是建立对应实例及其友元间的友好关系。例如，我们的 Blob 类应该将 BlobPtr 类和一个模板版本的 Blob 相

等运算符（最初是在 14.3.1 节（第 498 页）练习中为 StrBlob 定义的）定义为友元。

为了引用（类或函数）模板的一个特定实例，我们必须首先声明模板自身。一个模板声明包括模板参数列表：

665

```
// 前置声明，在 Blob 中声明友元所需要的
template <typename> class BlobPtr;
template <typename> class Blob; // 运算符==中的参数所需要的
template <typename T>
    bool operator==(const Blob<T>&, const Blob<T>&);

template <typename T> class Blob {
    // 每个 Blob 实例将访问权限授予用相同类型实例化的 BlobPtr 和相等运算符
    friend class BlobPtr<T>;
    friend bool operator==<T>
            (const Blob<T>&, const Blob<T>&);
    // 其他成员定义，与 12.1.1（第 405 页）相同
};
```

我们首先将 Blob、BlobPtr 和 operator==声明为模板。这些声明是 operator==函数的参数声明以及 Blob 中的友元声明所需要的。

友元的声明用 Blob 的模板形参作为它们自己的模板实参。因此，友好关系被限定在用相同类型实例化的 Blob 与 BlobPtr 相等运算符之间：

```
Blob<char> ca;    // BlobPtr<char>和 operator==<char>都是本对象的友元
Blob<int> ia;     // BlobPtr<int>和 operator==<int>都是本对象的友元
```

BlobPtr<char>的成员可以访问 ca（或任何其他 Blob<char>对象）的非 public 部分，但 ca 对 ia（或任何其他 Blob<int>对象）或 Blob 的任何其他实例都没有特殊访问权限。

通用和特定的模板友好关系

一个类也可以将另一个模板的每个实例都声明为自己的友元，或者限定特定的实例为友元：

```
// 前置声明，在将模板的一个特定实例声明为友元时要用到
template <typename T> class Pal;
class C { // C 是一个普通的非模板类
    friend class Pal<C>; // 用类 C 实例化的 Pal 是 C 的一个友元
    // Pal2 的所有实例都是 C 的友元；这种情况无须前置声明
    template <typename T> friend class Pal2;
};
template <typename T> class C2 { // C2 本身是一个类模板
    // C2 的每个实例将相同实例化的 Pal 声明为友元
    friend class Pal<T>; // Pal 的模板声明必须在作用域之内
    // Pal2 的所有实例都是 C2 的每个实例的友元，不需要前置声明
    template <typename X> friend class Pal2;
    // Pal3 是一个非模板类，它是 C2 所有实例的友元
    friend class Pal3; // 不需要 Pal3 的前置声明
};
```

为了让所有实例成为友元，友元声明中必须使用与类模板本身不同的模板参数。

666 令模板自己的类型参数成为友元

C++ 11

在新标准中，我们可以将模板类型参数声明为友元：

```
template <typename Type> class Bar {
friend Type; // 将访问权限授予用来实例化 Bar 的类型
    //...
};
```

此处我们将用来实例化 Bar 的类型声明为友元。因此，对于某个类型名 Foo，Foo 将成为 Bar<Foo>的友元，Sales_data 将成为 Bar<Sales_data>的友元，依此类推。

值得注意的是，虽然友元通常来说应该是一个类或是一个函数，但我们完全可以用一个内置类型来实例化 Bar。这种与内置类型的友好关系是允许的，以便我们能用内置类型来实例化 Bar 这样的类。

模板类型别名

类模板的一个实例定义了一个类类型，与任何其他类类型一样，我们可以定义一个 typedef（参见 2.5.1 节，第 60 页）来引用实例化的类：

```
typedef Blob<string> StrBlob;
```

这条 typedef 语句允许我们运行在 12.1.1 节（第 405 页）中编写的代码，而使用的却是用 string 实例化的模板版本的 Blob。由于模板不是一个类型，我们不能定义一个 typedef 引用一个模板。即，无法定义一个 typedef 引用 Blob<T>。

C++ 11

但是，新标准允许我们为类模板定义一个类型别名：

```
template<typename T> using twin = pair<T, T>;
twin<string> authors; // authors 是一个 pair<string, string>
```

在这段代码中，我们将 twin 定义为成员类型相同的 pair 的别名。这样，twin 的用户只需指定一次类型。

一个模板类型别名是一族类的别名：

```
twin<int> win_loss;  // win_loss 是一个 pair<int, int>
twin<double> area;   // area 是一个 pair<double, double>
```

就像使用类模板一样，当我们使用 twin 时，需要指出希望使用哪种特定类型的 twin。

当我们定义一个模板类型别名时，可以固定一个或多个模板参数：

```
template <typename T> using partNo = pair<T, unsigned>;
partNo<string> books; // books 是一个 pair<string, unsigned>
partNo<Vehicle> cars; // cars 是一个 pair<Vehicle, unsigned>
partNo<Student> kids; // kids 是一个 pair<Student, unsigned>
```

这段代码中我们将 partNo 定义为一族类型的别名，这族类型是 second 成员为 unsigned 的 pair。partNo 的用户需要指出 pair 的 first 成员的类型，但不能指定 second 成员的类型。

667 类模板的 static 成员

与任何其他类相同，类模板可以声明 static 成员（参见 7.6 节，第 269 页）：

```
template <typename T> class Foo {
public:
```

```
    static std::size_t count() { return ctr; }
    // 其他接口成员
private:
    static std::size_t ctr;
    // 其他实现成员
};
```

在这段代码中，Foo 是一个类模板，它有一个名为 count 的 public static 成员函数和一个名为 ctr 的 private static 数据成员。每个 Foo 的实例都有其自己的 static 成员实例。即，对任意给定类型 X，都有一个 Foo<X>::ctr 和一个 Foo<X>::count 成员。所有 Foo<X>类型的对象共享相同的 ctr 对象和 count 函数。例如，

```
// 实例化 static 成员 Foo<string>::ctr 和 Foo<string>::count
Foo<string> fs;
// 所有三个对象共享相同的 Foo<int>::ctr 和 Foo<int>::count 成员
Foo<int> fi, fi2, fi3;
```

与任何其他 static 数据成员相同，模板类的每个 static 数据成员必须有且仅有一个定义。但是，类模板的每个实例都有一个独有的 static 对象。因此，与定义模板的成员函数类似，我们将 static 数据成员也定义为模板：

```
template <typename T>
size_t Foo<T>::ctr = 0; // 定义并初始化 ctr
```

与类模板的其他任何成员类似，定义的开始部分是模板参数列表，随后是我们定义的成员的类型和名字。与往常一样，成员名包括成员的类名，对于从模板生成的类来说，类名包括模板实参。因此，当使用一个特定的模板实参类型实例化 Foo 时，将会为该类类型实例化一个独立的 ctr，并将其初始化为 0。

与非模板类的静态成员相同，我们可以通过类类型对象来访问一个类模板的 static 成员，也可以使用作用域运算符直接访问成员。当然，为了通过类来直接访问 static 成员，我们必须引用一个特定的实例：

```
Foo<int> fi;                    // 实例化 Foo<int>类和 static 数据成员 ctr
auto ct = Foo<int>::count();    // 实例化 Foo<int>::count
ct = fi.count();                // 使用 Foo<int>::count
ct = Foo::count();              // 错误：使用哪个模板实例的 count?
```

类似任何其他成员函数，一个 static 成员函数只有在使用时才会实例化。

16.1.2 节练习

668

练习 16.9：什么是函数模板？什么是类模板？

练习 16.10：当一个类模板被实例化时，会发生什么？

练习 16.11：下面 List 的定义是错误的。应如何修正它？

```
template <typename elemType> class ListItem;
template <typename elemType> class List {
public:
    List<elemType>();
    List<elemType>(const List<elemType> &);
    List<elemType>& operator=(const List<elemType> &);
    ~List();
```

```
        void insert(ListItem *ptr, elemType value);
    private:
        ListItem *front, *end;
    };
```

练习 16.12：编写你自己版本的 Blob 和 BlobPtr 模板，包含书中未定义的多个 const 成员。

练习 16.13：解释你为 BlobPtr 的相等和关系运算符选择哪种类型的友好关系？

练习 16.14：编写 Screen 类模板，用非类型参数定义 Screen 的高和宽。

练习 16.15：为你的 Screen 模板实现输入和输出运算符。Screen 类需要哪些友元（如果需要的话）来令输入和输出运算符正确工作？解释每个友元声明（如果有的话）为什么是必要的。

练习 16.16：将 StrVec 类（参见 13.5 节，第 465 页）重写为模板，命名为 Vec。

16.1.3 模板参数

类似函数参数的名字，一个模板参数的名字也没有什么内在含义。我们通常将类型参数命名为 T，但实际上我们可以使用任何名字：

```
template <typename Foo> Foo calc(const Foo& a, const Foo& b)
{
    Foo tmp = a; // tmp 的类型与参数和返回类型一样
    //...
    return tmp;  // 返回类型和参数类型一样
}
```

模板参数与作用域

模板参数遵循普通的作用域规则。一个模板参数名的可用范围是在其声明之后，至模板声明或定义结束之前。与任何其他名字一样，模板参数会隐藏外层作用域中声明的相同名字。但是，与大多数其他上下文不同，在模板内不能重用模板参数名：

669

```
typedef double A;
template <typename A, typename B> void f(A a, B b)
{
    A tmp = a;    // tmp 的类型为模板参数 A 的类型，而非 double
    double B;     // 错误：重声明模板参数 B
}
```

正常的名字隐藏规则决定了 A 的 typedef 被类型参数 A 隐藏。因此，tmp 不是一个 double，其类型是使用 f 时绑定到类型参数 A 的类型。由于我们不能重用模板参数名，声明名字为 B 的变量是错误的。

由于参数名不能重用，所以一个模板参数名在一个特定模板参数列表中只能出现一次：

```
// 错误：非法重用模板参数名 V
template <typename V, typename V> //...
```

模板声明

模板声明必须包含模板参数：

```
// 声明但不定义 compare 和 Blob
template <typename T> int compare(const T&, const T&);
template <typename T> class Blob;
```

与函数参数相同，声明中的模板参数的名字不必与定义中相同：

```
// 3 个 calc 都指向相同的函数模板
template <typename T> T calc(const T&, const T&); // 声明
template <typename U> U calc(const U&, const U&); // 声明
// 模板的定义
template <typename Type>
Type calc(const Type& a, const Type& b) { /* ... */ }
```

当然，一个给定模板的每个声明和定义必须有相同数量和种类（即，类型或非类型）的参数。

一个特定文件所需要的所有模板的声明通常一起放置在文件开始位置，出现于任何使用这些模板的代码之前，原因我们将在 16.3 节（第 617 页）中解释。

使用类的类型成员

回忆一下，我们用作用域运算符（::）来访问 static 成员和类型成员（参见 7.4 节，第 253 页和 7.6 节，第 269 页）。在普通（非模板）代码中，编译器掌握类的定义。因此，它知道通过作用域运算符访问的名字是类型还是 static 成员。例如，如果我们写下 string::size_type，编译器有 string 的定义，从而知道 size_type 是一个类型。 670

但对于模板代码就存在困难。例如，假定 T 是一个模板类型参数，当编译器遇到类似 T::mem 这样的代码时，它不会知道 mem 是一个类型成员还是一个 static 数据成员，直至实例化时才会知道。但是，为了处理模板，编译器必须知道名字是否表示一个类型。例如，假定 T 是一个类型参数的名字，当编译器遇到如下形式的语句时：

```
T::size_type * p;
```

它需要知道我们是正在定义一个名为 p 的变量还是将一个名为 size_type 的 static 数据成员与名为 p 的变量相乘。

默认情况下，C++语言假定通过作用域运算符访问的名字不是类型。因此，如果我们希望使用一个模板类型参数的类型成员，就必须显式告诉编译器该名字是一个类型。我们通过使用关键字 typename 来实现这一点：

```
template <typename T>
typename T::value_type top(const T& c)
{
    if (!c.empty())
        return c.back();
    else
        return typename T::value_type();
}
```

我们的 top 函数期待一个容器类型的实参，它使用 typename 指明其返回类型并在 c 中没有元素时生成一个值初始化的元素（参见 7.5.3 节，第 262 页）返回给调用者。

当我们希望通知编译器一个名字表示类型时，必须使用关键字 typename，而不能使用 class。

默认模板实参

C++ 11

就像我们能为函数参数提供默认实参一样（参见 6.5.1 节，第 211 页），我们也可以提供**默认模板实参**（default template argument）。在新标准中，我们可以为函数和类模板提供默认实参。而更早的 C++标准只允许为类模板提供默认实参。

例如，我们重写 compare，默认使用标准库的 less 函数对象模板（参见 14.8.2 节，第 509 页）：

```
// compare 有一个默认模板实参 less<T>和一个默认函数实参 F()
template <typename T, typename F = less<T>>
int compare(const T &v1, const T &v2, F f = F())
{
    if (f(v1, v2)) return -1;
    if (f(v2, v1)) return 1;
    return 0;
}
```

671 在这段代码中，我们为模板添加了第二个类型参数，名为 F，表示可调用对象（参见 10.3.2 节，第 346 页）的类型；并定义了一个新的函数参数 f，绑定到一个可调用对象上。

我们为此模板参数提供了默认实参，并为其对应的函数参数也提供了默认实参。默认模板实参指出 compare 将使用标准库的 less 函数对象类，它是使用与 compare 一样的类型参数实例化的。默认函数实参指出 f 将是类型 F 的一个默认初始化的对象。

当用户调用这个版本的 compare 时，可以提供自己的比较操作，但这并不是必需的：

```
bool i = compare(0, 42); // 使用 less; i 为-1
// 结果依赖于 item1 和 item2 中的 isbn
Sales_data item1(cin), item2(cin);
bool j = compare(item1, item2, compareIsbn);
```

第一个调用使用默认函数实参，即，类型 less<T>的一个默认初始化对象。在此调用中，T 为 int，因此可调用对象的类型为 less<int>。compare 的这个实例化版本将使用 less<int>进行比较操作。

在第二个调用中，我们传递给 compare 三个实参：compareIsbn（参见 11.2.2 节，第 379 页）和两个 Sales_data 类型的对象。当传递给 compare 三个实参时，第三个实参的类型必须是一个可调用对象，该可调用对象的返回类型必须能转换为 bool 值，且接受的实参类型必须与 compare 的前两个实参的类型兼容。与往常一样，模板参数的类型从它们对应的函数实参推断而来。在此调用中，T 的类型被推断为 Sales_data，F 被推断为 compareIsbn 的类型。

与函数默认实参一样，对于一个模板参数，只有当它右侧的所有参数都有默认实参时，它才可以有默认实参。

模板默认实参与类模板

无论何时使用一个类模板，我们都必须在模板名之后接上尖括号。尖括号指出类必须从一个模板实例化而来。特别是，如果一个类模板为其所有模板参数都提供了默认实参，且我们希望使用这些默认实参，就必须在模板名之后跟一个空尖括号对：

```
template <class T = int> class Numbers { // T 默认为 int
public:
    Numbers(T v = 0): val(v) { }
```

```
    // 对数值的各种操作
private:
    T val;
};
Numbers<long double> lots_of_precision;
Numbers<> average_precision; // 空<>表示我们希望使用默认类型
```

此例中我们实例化了两个 Numbers 版本：average_precision 是用 int 代替 T 实例化得到的；lots_of_precision 是用 long double 代替 T 实例化而得到的。

16.1.3 节练习

672

练习 16.17：声明为 typename 的类型参数和声明为 class 的类型参数有什么不同（如果有的话）？什么时候必须使用 typename？

练习 16.18：解释下面每个函数模板声明并指出它们是否非法。更正你发现的每个错误。

```
(a) template <typename T, U, typename V> void f1(T, U, V);
(b) template <typename T> T f2(int &T);
(c) inline template <typename T> T foo(T, unsigned int*);
(d) template <typename T> f4(T, T);
(e) typedef char Ctype;
    template <typename Ctype> Ctype f5(Ctype a);
```

练习 16.19：编写函数，接受一个容器的引用，打印容器中的元素。使用容器的 size_type 和 size 成员来控制打印元素的循环。

练习 16.20：重写上一题的函数，使用 begin 和 end 返回的迭代器来控制循环。

16.1.4 成员模板

一个类（无论是普通类还是类模板）可以包含本身是模板的成员函数。这种成员被称为**成员模板**（member template）。成员模板不能是虚函数。

普通（非模板）类的成员模板

作为普通类包含成员模板的例子，我们定义一个类，类似 unique_ptr 所使用的默认删除器类型（参见 12.1.5 节，第 418 页）。类似默认删除器，我们的类将包含一个重载的函数调用运算符（参见 14.8 节，第 506 页），它接受一个指针并对此指针执行 delete。与默认删除器不同，我们的类还将在删除器被执行时打印一条信息。由于希望删除器适用于任何类型，所以我们将调用运算符定义为一个模板：

```
// 函数对象类，对给定指针执行 delete
class DebugDelete {
public:
    DebugDelete(std::ostream &s = std::cerr): os(s) { }
    // 与任何函数模板相同，T 的类型由编译器推断
    template <typename T> void operator()(T *p) const
      { os << "deleting unique_ptr" << std::endl; delete p; }
private:
    std::ostream &os;
};
```

673 与任何其他模板相同，成员模板也是以模板参数列表开始的。每个 DebugDelete 对象都有一个 ostream 成员，用于写入数据；还包含一个自身是模板的成员函数。我们可以用这个类代替 delete：

```
double* p = new double;
DebugDelete d; // 可像 delete 表达式一样使用的对象
d(p); // 调用 DebugDelete::operator()(double*)，释放 p
int* ip = new int;
// 在一个临时 DebugDelete 对象上调用 operator()(int*)
DebugDelete()(ip);
```

由于调用一个 DebugDelete 对象会 delete 其给定的指针，我们也可以将 DebugDelete 用作 unique_ptr 的删除器。为了重载 unique_ptr 的删除器，我们在尖括号内给出删除器类型，并提供一个这种类型的对象给 unique_ptr 的构造函数（参见 12.1.5 节，第 418 页）：

```
// 销毁 p 指向的对象
// 实例化 DebugDelete::operator()<int>(int *)
unique_ptr<int, DebugDelete> p(new int, DebugDelete());
// 销毁 sp 指向的对象
// 实例化 DebugDelete::operator()<string>(string*)
unique_ptr<string, DebugDelete> sp(new string, DebugDelete());
```

在本例中，我们声明 p 的删除器的类型为 DebugDelete，并在 p 的构造函数中提供了该类型的一个未命名对象。

unique_ptr 的析构函数会调用 DebugDelete 的调用运算符。因此，无论何时 unique_ptr 的析构函数实例化时，DebugDelete 的调用运算符都会实例化：因此，上述定义会这样实例化.

```
// DebugDelete 的成员模板实例化样例
void DebugDelete::operator()(int *p) const { delete p; }
void DebugDelete::operator()(string *p) const { delete p; }
```

类模板的成员模板

对于类模板，我们也可以为其定义成员模板。在此情况下，类和成员各自有自己的、独立的模板参数。

例如，我们将为 Blob 类定义一个构造函数，它接受两个迭代器，表示要拷贝的元素范围。由于我们希望支持不同类型序列的迭代器，因此将构造函数定义为模板：

```
template <typename T> class Blob {
    template <typename It> Blob(It b, It e);
    //...
};
```

此构造函数有自己的模板类型参数 It，作为它的两个函数参数的类型。

与类模板的普通函数成员不同，成员模板是函数模板。当我们在类模板外定义一个成
674 员模板时，必须同时为类模板和成员模板提供模板参数列表。类模板的参数列表在前，后跟成员自己的模板参数列表：

```
template <typename T>     // 类的类型参数
template <typename It>    // 构造函数的类型参数
    Blob<T>::Blob(It b, It e):
```

```
data(std::make_shared<std::vector<T>>(b, e)) { }
```

在此例中，我们定义了一个类模板的成员，类模板有一个模板类型参数，命名为 T。而成员自身是一个函数模板，它有一个名为 It 的类型参数。

实例化与成员模板

为了实例化一个类模板的成员模板，我们必须同时提供类和函数模板的实参。与往常一样，我们在哪个对象上调用成员模板，编译器就根据该对象的类型来推断类模板参数的实参。与普通函数模板相同，编译器通常根据传递给成员模板的函数实参来推断它的模板实参（参见 16.1.1 节，第 579 页）：

```
int ia[] = {0,1,2,3,4,5,6,7,8,9};
vector<long> vi = {0,1,2,3,4,5,6,7,8,9};
list<const char*> w = {"now", "is", "the", "time"};
// 实例化 Blob<int>类及其接受两个 int*参数的构造函数
Blob<int> a1(begin(ia), end(ia));
// 实例化 Blob<int>类的接受两个 vector<long>::iterator 的构造函数
Blob<int> a2(vi.begin(), vi.end());
// 实例化 Blob<string>及其接受两个 list<const char*>::iterator 参数的构造函数
Blob<string> a3(w.begin(), w.end());
```

当我们定义 a1 时，显式地指出编译器应该实例化一个 int 版本的 Blob。构造函数自己的类型参数则通过 begin(ia) 和 end(ia) 的类型来推断，结果为 int*。因此，a1 的定义实例化了如下版本：

```
Blob<int>::Blob(int*, int*);
```

a2 的定义使用了已经实例化了的 Blob<int>类，并用 vector<short>::iterator 替换 It 来实例化构造函数。a3 的定义（显式地）实例化了一个 string 版本的 Blob，并（隐式地）实例化了该类的成员模板构造函数，其模板参数被绑定到 list<const char*>。

16.1.4 节练习

675

练习 16.21：编写你自己的 DebugDelete 版本。

练习 16.22：修改 12.3 节（第 430 页）中你的 TextQuery 程序，令 shared_ptr 成员使用 DebugDelete 作为它们的删除器（参见 12.1.4 节，第 415 页）。

练习 16.23：预测在你的查询主程序中何时会执行调用运算符。如果你的预测和实际不符，确认你理解了原因。

练习 16.24：为你的 Blob 模板添加一个构造函数，它接受两个迭代器。

16.1.5 控制实例化

当模板被使用时才会进行实例化（参见 16.1.1 节，第 582 页）这一特性意味着，相同的实例可能出现在多个对象文件中。当两个或多个独立编译的源文件使用了相同的模板，并提供了相同的模板参数时，每个文件中就都会有该模板的一个实例。

在大系统中，在多个文件中实例化相同模板的额外开销可能非常严重。在新标准中，我们可以通过**显式实例化**（explicit instantiation）来避免这种开销。一个显式实例化有如下

形式：

```
extern template declaration;      // 实例化声明
template declaration;             // 实例化定义
```

declaration 是一个类或函数声明，其中所有模板参数已被替换为模板实参。例如，

```
// 实例化声明与定义
extern template class Blob<string>;              // 声明
template int compare(const int&, const int&); // 定义
```

当编译器遇到 extern 模板声明时，它不会在本文件中生成实例化代码。将一个实例化声明为 extern 就表示承诺在程序其他位置有该实例化的一个非 extern 声明（定义）。对于一个给定的实例化版本，可能有多个 extern 声明，但必须只有一个定义。

由于编译器在使用一个模板时自动对其实例化，因此 extern 声明必须出现在任何使用此实例化版本的代码之前：

```
// Application.cc
// 这些模板类型必须在程序其他位置进行实例化
extern template class Blob<string>;
extern template int compare(const int&, const int&);
Blob<string> sa1, sa2; // 实例化会出现在其他位置
// Blob<int>及其接受 initializer_list 的构造函数在本文件中实例化
Blob<int> a1 = {0,1,2,3,4,5,6,7,8,9};
Blob<int> a2(a1); // 拷贝构造函数在本文件中实例化
int i = compare(a1[0], a2[0]); // 实例化出现在其他位置
```

676 文件 Application.o 将包含 Blob<int>的实例及其接受 initializer_list 参数的构造函数和拷贝构造函数的实例。而 compare<int>函数和 Blob<string>类将不在本文件中进行实例化。这些模板的定义必须出现在程序的其他文件中：

```
// templateBuild.cc
// 实例化文件必须为每个在其他文件中声明为 extern 的类型和函数提供一个（非 extern）
// 的定义
template int compare(const int&, const int&);
template class Blob<string>; // 实例化类模板的所有成员
```

当编译器遇到一个实例化定义（与声明相对）时，它为其生成代码。因此，文件 templateBuild.o 将会包含 compare 的 int 实例化版本的定义和 Blob<string>类的定义。当我们编译此应用程序时，必须将 templateBuild.o 和 Application.o 链接到一起。

WARNING

对每个实例化声明，在程序中某个位置必须有其显式的实例化定义。

实例化定义会实例化所有成员

一个类模板的实例化定义会实例化该模板的所有成员，包括内联的成员函数。当编译器遇到一个实例化定义时，它不了解程序使用哪些成员函数。因此，与处理类模板的普通实例化不同，编译器会实例化该类的所有成员。即使我们不使用某个成员，它也会被实例化。因此，我们用来显式实例化一个类模板的类型，必须能用于模板的所有成员。

在一个类模板的实例化定义中，所用类型必须能用于模板的所有成员函数。

16.1.5 节练习

练习 16.25：解释下面这些声明的含义：

```
extern template class vector<string>;
template class vector<Sales_data>;
```

练习 16.26：假设 NoDefault 是一个没有默认构造函数的类，我们可以显式实例化 vector<NoDefault>吗？如果不可以，解释为什么。

练习 16.27：对下面每条带标签的语句，解释发生了什么样的实例化（如果有的话）。如果一个模板被实例化，解释为什么；如果未实例化，解释为什么没有。

```
template <typename T> class Stack { };
void f1(Stack<char>);                    // (a)
class Exercise {
    Stack<double> &rsd;                  // (b)
    Stack<int> si;                       // (c)
};
int main() {
    Stack<char> *sc;                     // (d)
    f1(*sc);                             // (e)
    int iObj = sizeof(Stack< string >); // (f)
}
```

16.1.6 效率与灵活性

对模板设计者所面对的设计选择，标准库智能指针类型（参见 12.1 节，第 400 页）给出了一个很好的展示。

shared_ptr 和 unique_ptr 之间的明显不同是它们管理所保存的指针的策略——前者给予我们共享指针所有权的能力；后者则独占指针。这一差异对两个类的功能来说是至关重要的。

这两个类的另一个差异是它们允许用户重载默认删除器的方式。我们可以很容易地重载一个 shared_ptr 的删除器，只要在创建或 reset 指针时传递给它一个可调用对象即可。与之相反，删除器的类型是一个 unique_ptr 对象的类型的一部分。用户必须在定义 unique_ptr 时以显式模板实参的形式提供删除器的类型。因此，对于 unique_ptr 的用户来说，提供自己的删除器就更为复杂。

如何处理删除器的差异实际上就是这两个类功能的差异。但是，如我们将要看到的，这一实现策略上的差异可能对性能有重要影响。 677

在运行时绑定删除器

虽然我们不知道标准库类型是如何实现的，但可以推断出，shared_ptr 必须能直接访问其删除器。即，删除器必须保存为一个指针或一个封装了指针的类（如 function，参见 14.8.3 节，第 512 页）。

我们可以确定 shared_ptr 不是将删除器直接保存为一个成员，因为删除器的类型

直到运行时才会知道。实际上，在一个 shared_ptr 的生存期中，我们可以随时改变其删除器的类型。我们可以使用一种类型的删除器构造一个 shared_ptr，随后使用 reset 赋予此 shared_ptr 另一种类型的删除器。通常，类成员的类型在运行时是不能改变的。因此，不能直接保存删除器。

为了考察删除器是如何正确工作的，让我们假定 shared_ptr 将它管理的指针保存在一个成员 p 中，且删除器是通过一个名为 del 的成员来访问的。则 shared_ptr 的析构函数必须包含类似下面这样的语句：

```
// del 的值只有在运行时才知道；通过一个指针来调用它
del ? del(p) : delete p; // del(p)需要运行时跳转到 del 的地址
```

678 由于删除器是间接保存的，调用 del(p)需要一次运行时的跳转操作，转到 del 中保存的地址来执行对应的代码。

在编译时绑定删除器

现在，让我们来考察 unique_ptr 可能的工作方式。在这个类中，删除器的类型是类类型的一部分。即，unique_ptr 有两个模板参数，一个表示它所管理的指针，另一个表示删除器的类型。由于删除器的类型是 unique_ptr 类型的一部分，因此删除器成员的类型在编译时是知道的，从而删除器可以直接保存在 unique_ptr 对象中。

unique_ptr 的析构函数与 shared_ptr 的析构函数类似，也是对其保存的指针调用用户提供的删除器或执行 delete:

```
// del 在编译时绑定；直接调用实例化的删除器
del(p); // 无运行时额外开销
```

del 的类型或者是默认删除器类型，或者是用户提供的类型。到底是哪种情况没有关系，应该执行的代码在编译时肯定会知道。实际上，如果删除器是类似 DebugDelete（参见 16.1.4 节，第 595 页）之类的东西，这个调用甚至可能被编译为内联形式。

通过在编译时绑定删除器，unique_ptr 避免了间接调用删除器的运行时开销。通过在运行时绑定删除器，shared_ptr 使用户重载删除器更为方便。

16.1.6 节练习

练习 16.28：编写你自己版本的 shared_ptr 和 unique_ptr。

练习 16.29：修改你的 Blob 类，用你自己的 shared_ptr 代替标准库中的版本。

练习 16.30：重新运行你的一些程序，验证你的 shared_ptr 类和修改后的 Blob 类。（注意：实现 weak_ptr 类型超出了本书范围，因此你不能将 BlobPtr 类与你修改后的 Blob 一起使用。）

练习 16.31：如果我们将 DebugDelete 与 unique_ptr 一起使用，解释编译器将删除器处理为内联形式的可能方式。

16.2 模板实参推断

我们已经看到，对于函数模板，编译器利用调用中的函数实参来确定其模板参数。从函数实参来确定模板实参的过程被称为**模板实参推断**（template argument deduction）。在模

板实参推断过程中，编译器使用函数调用中的实参类型来寻找模板实参，用这些模板实参生成的函数版本与给定的函数调用最为匹配。

16.2.1 类型转换与模板类型参数

679

与非模板函数一样，我们在一次调用中传递给函数模板的实参被用来初始化函数的形参。如果一个函数形参的类型使用了模板类型参数，那么它采用特殊的初始化规则。只有很有限的几种类型转换会自动地应用于这些实参。编译器通常不是对实参进行类型转换，而是生成一个新的模板实例。

与往常一样，顶层 const（参见 2.4.3 节，第 57 页）无论是在形参中还是在实参中，都会被忽略。在其他类型转换中，能在调用中应用于函数模板的包括如下两项。

- const 转换：可以将一个非 const 对象的引用（或指针）传递给一个 const 的引用（或指针）形参（参见 4.11.2 节，第 144 页）。
- 数组或函数指针转换：如果函数形参不是引用类型，则可以对数组或函数类型的实参应用正常的指针转换。一个数组实参可以转换为一个指向其首元素的指针。类似的，一个函数实参可以转换为一个该函数类型的指针（参见 4.11.2 节，第 143 页）。

其他类型转换，如算术转换（参见 4.11.1 节，第 142 页）、派生类向基类的转换（参见 15.2.2 节，第 530 页）以及用户定义的转换（参见 7.5.4 节，第 263 页和 14.9 节，第 514 页），都不能应用于函数模板。

作为一个例子，考虑对函数 fobj 和 fref 的调用。fobj 函数拷贝它的参数，而 fref 的参数是引用类型：

```
template <typename T> T fobj(T, T); // 实参被拷贝
template <typename T> T fref(const T&, const T&); // 引用
string s1("a value");
const string s2("another value");
fobj(s1, s2);     // 调用 fobj(string, string); const 被忽略
fref(s1, s2);     // 调用 fref(const string&, const string&)
                  // 将 s1 转换为 const 是允许的
int a[10], b[42];
fobj(a, b);       // 调用 f(int*, int*)
fref(a, b);       // 错误：数组类型不匹配
```

在第一对调用中，我们传递了一个 string 和一个 const string。虽然这些类型不严格匹配，但两个调用都是合法的。在 fobj 调用中，实参被拷贝，因此原对象是否是 const 没有关系。在 fref 调用中，参数类型是 const 的引用。对于一个引用参数来说，转换为 const 是允许的，因此这个调用也是合法的。

在下一对调用中，我们传递了数组实参，两个数组大小不同，因此是不同类型。在 fobj 调用中，数组大小不同无关紧要。两个数组都被转换为指针。fobj 中的模板类型为 int*。但是，fref 调用是不合法的。如果形参是一个引用，则数组不会转换为指针（参见 6.2.4 节，第 195 页）。a 和 b 的类型是不匹配的，因此调用是错误的。

将实参传递给带模板类型的函数形参时，能够自动应用的类型转换只有 const 转换及数组或函数到指针的转换。

680

使用相同模板参数类型的函数形参

一个模板类型参数可以用作多个函数形参的类型。由于只允许有限的几种类型转换，因此传递给这些形参的实参必须具有相同的类型。如果推断出的类型不匹配，则调用就是错误的。例如，我们的 compare 函数（参见 16.1.1 节，第 578 页）接受两个 const T& 参数，其实参必须是相同类型：

```
long lng;
compare(lng, 1024); // 错误：不能实例化 compare(long, int)
```

此调用是错误的，因为传递给 compare 的实参类型不同。从第一个函数实参推断出的模板实参为 long，从第二个函数实参推断出的模板实参为 int。这些类型不匹配，因此模板实参推断失败。

如果希望允许对函数实参进行正常的类型转换，我们可以将函数模板定义为两个类型参数：

```
// 实参类型可以不同，但必须兼容
template <typename A, typename B>
int flexibleCompare(const A& v1, const B& v2)
{
    if (v1 < v2) return -1;
    if (v2 < v1) return 1;
    return 0;
}
```

现在用户可以提供不同类型的实参了：

```
long lng;
flexibleCompare(lng, 1024); // 正确:调用 flexibleCompare(long, int)
```

当然，必须定义了能比较这些类型的值的<运算符。

正常类型转换应用于普通函数实参

函数模板可以有用普通类型定义的参数，即，不涉及模板类型参数的类型。这种函数实参不进行特殊处理；它们正常转换为对应形参的类型（参见 6.1 节，第 183 页）。例如，考虑下面的模板：

```
template <typename T> ostream &print(ostream &os, const T &obj)
{
    return os << obj;
}
```

第一个函数参数是一个已知类型 ostream&。第二个参数 obj 则是模板参数类型。由于 os 的类型是固定的，因此当调用 print 时，传递给它的实参会进行正常的类型转换：

681

```
print(cout, 42); // 实例化 print(ostream&, int)
ofstream f("output");
print(f, 10);    // 使用 print(ostream&, int)；将 f 转换为 ostream&
```

在第一个调用中，第一个实参的类型严格匹配第一个参数的类型。此调用会实例化接受一个 ostream&和一个 int 的 print 版本。在第二个调用中，第一个实参是一个 ofstream，它可以转换为 ostream&（参见 8.2.1 节，第 284 页）。由于此参数的类型不依赖于模板参数，因此编译器会将 f 隐式转换为 ostream&。

如果函数参数类型不是模板参数，则对实参进行正常的类型转换。

16.2.1 节练习

练习 16.32：在模板实参推断过程中发生了什么？

练习 16.33：指出在模板实参推断过程中允许对函数实参进行的两种类型转换。

练习 16.34：对下面的代码解释每个调用是否合法。如果合法，T 的类型是什么？如果不合法，为什么？

```
template <class T> int compare(const T&, const T&);
(a) compare("hi", "world"); (b) compare("bye", "dad");
```

练习 16.35：下面调用中哪些是错误的（如果有的话）？如果调用合法，T 的类型是什么？如果调用不合法，问题何在？

```
template <typename T> T calc(T, int);
template <typename T> T fcn(T, T);
double d; float f; char c;
(a) calc(c, 'c');          (b) calc(d, f);
(c) fcn(c, 'c');           (d) fcn(d, f);
```

练习 16.36：进行下面的调用会发生什么：

```
template <typename T> f1(T, T);
template <typename T1, typename T2) f2(T1, T2);
int i = 0, j = 42, *p1 = &i, *p2 = &j;
const int *cp1 = &i, *cp2 = &j;
(a) f1(p1, p2);       (b) f2(p1, p2);   (c) f1(cp1, cp2);
(d) f2(cp1, cp2);     (e) f1(p1, cp1);  (e) f2(p1, cp1);
```

16.2.2 函数模板显式实参

在某些情况下，编译器无法推断出模板实参的类型。其他一些情况下，我们希望允许用户控制模板实例化。当函数返回类型与参数列表中任何类型都不相同时，这两种情况最常出现。 682

指定显式模板实参

作为一个允许用户指定使用类型的例子，我们将定义一个名为 sum 的函数模板，它接受两个不同类型的参数。我们希望允许用户指定结果的类型。这样，用户就可以选择合适的精度。

我们可以定义表示返回类型的第三个模板参数，从而允许用户控制返回类型：

```
// 编译器无法推断 T1，它未出现在函数参数列表中
template <typename T1, typename T2, typename T3>
T1 sum(T2, T3);
```

在本例中，没有任何函数实参的类型可用来推断 T1 的类型。每次调用 sum 时调用者都必须为 T1 提供一个**显式模板实参**（explicit template argument）。

我们提供显式模板实参的方式与定义类模板实例的方式相同。显式模板实参在尖括号中给出，位于函数名之后，实参列表之前：

```
// T1 是显式指定的，T2 和 T3 是从函数实参类型推断而来的
auto val3 = sum<long long>(i, lng); // long long sum(int, long)
```

此调用显式指定 T1 的类型。而 T2 和 T3 的类型则由编译器从 i 和 lng 的类型推断出来。

显式模板实参按由左至右的顺序与对应的模板参数匹配；第一个模板实参与第一个模板参数匹配，第二个实参与第二个参数匹配，依此类推。只有尾部（最右）参数的显式模板实参才可以忽略，而且前提是它们可以从函数参数推断出来。如果我们的 sum 函数按照如下形式编写：

```
// 糟糕的设计：用户必须指定所有三个模板参数
template <typename T1, typename T2, typename T3>
T3 alternative_sum(T2, T1);
```

则我们总是必须为所有三个形参指定实参：

```
// 错误：不能推断前几个模板参数
auto val3 = alternative_sum<long long>(i, lng);
// 正确：显式指定了所有三个参数
auto val2 = alternative_sum<long long, int, long>(i, lng);
```

正常类型转换应用于显式指定的实参

对于用普通类型定义的函数参数，允许进行正常的类型转换（参见 16.2.1 节，第 602 页），出于同样的原因，对于模板类型参数已经显式指定了的函数实参，也进行正常的类型转换：

683

```
long lng;
compare(lng, 1024);            // 错误：模板参数不匹配
compare<long>(lng, 1024);      // 正确：实例化 compare(long, long)
compare<int>(lng, 1024);       // 正确：实例化 compare(int, int)
```

如我们所见，第一个调用是错误的，因为传递给 compare 的实参必须具有相同的类型。如果我们显式指定模板类型参数，就可以进行正常类型转换了。因此，调用 compare<long>等价于调用一个接受两个 const long&参数的函数。int 类型的参数被自动转化为 long。在第三个调用中，T 被显式指定为 int，因此 lng 被转换为 int。

16.2.2 节练习

练习 16.37：标准库 max 函数有两个参数，它返回实参中的较大者。此函数有一个模板类型参数。你能在调用 max 时传递给它一个 int 和一个 double 吗？如果可以，如何做？如果不可以，为什么？

练习 16.38：当我们调用 make_share（参见 12.1.1 节，第 401 页）时，必须提供一个显式模板实参。解释为什么需要显式模板实参以及它是如何使用的。

练习 16.39：对 16.1.1 节（第 578 页）中的原始版本的 compare 函数，使用一个显式模板实参，使得可以向函数传递两个字符串字面常量。

16.2.3 尾置返回类型与类型转换

当我们希望用户确定返回类型时，用显式模板实参表示模板函数的返回类型是很有效的。但在其他情况下，要求显式指定模板实参会给用户增添额外负担，而且不会带来什么好处。例如，我们可能希望编写一个函数，接受表示序列的一对迭代器和返回序列中一个

元素的引用：

```
template <typename It>
??? &fcn(It beg, It end)
{
    // 处理序列
    return *beg; // 返回序列中一个元素的引用
}
```

我们并不知道返回结果的准确类型，但知道所需类型是所处理的序列的元素类型：

```
vector<int> vi = {1,2,3,4,5};
Blob<string> ca = { "hi", "bye" };
auto &i = fcn(vi.begin(), vi.end()); // fcn 应该返回 int&
auto &s = fcn(ca.begin(), ca.end()); // fcn 应该返回 string&
```

此例中，我们知道函数应该返回*beg，而且知道我们可以用 decltype(*beg) 来获取此表达式的类型。但是，在编译器遇到函数的参数列表之前，beg 都是不存在的。为了定义此函数，我们必须使用尾置返回类型（参见 6.3.3 节，第 206 页）。由于尾置返回出现在参数列表之后，它可以使用函数的参数：

684

C++ 11

```
// 尾置返回允许我们在参数列表之后声明返回类型
template <typename It>
auto fcn(It beg, It end) -> decltype(*beg)
{
    // 处理序列
    return *beg; // 返回序列中一个元素的引用
}
```

此例中我们通知编译器 fcn 的返回类型与解引用 beg 参数的结果类型相同。解引用运算符返回一个左值（参见 4.1.1 节，第 121 页），因此通过 decltype 推断的类型为 beg 表示的元素的类型的引用。因此，如果对一个 string 序列调用 fcn，返回类型将是 string&。如果是 int 序列，则返回类型是 int&。

进行类型转换的标准库模板类

有时我们无法直接获得所需要的类型。例如，我们可能希望编写一个类似 fcn 的函数，但返回一个元素的值（参见 6.3.2 节，第 201 页）而非引用。

在编写这个函数的过程中，我们面临一个问题：对于传递的参数的类型，我们几乎一无所知。在此函数中，我们知道唯一可以使用的操作是迭代器操作，而所有迭代器操作都不会生成元素，只能生成元素的引用。

为了获得元素类型，我们可以使用标准库的**类型转换**（type transformation）模板。这些模板定义在头文件 type_traits 中。这个头文件中的类通常用于所谓的模板元程序设计，这一主题已超出本书的范围。但是，类型转换模板在普通编程中也很有用。表 16.1 列出了这些模板，我们将在 16.5 节（第 624 页）中看到它们是如何实现的。

在本例中，我们可以使用 remove_reference 来获得元素类型。remove_reference 模板有一个模板类型参数和一个名为 type 的（public）类型成员。如果我们用一个引用类型实例化 remove_reference，则 type 将表示被引用的类型。例如，如果我们实例化 remove_reference<int&>，则 type 成员将是 int。类似的，如果我们实例化 remove_reference<string&>，则 type 成员将是 string，依此类推。更一般的，给定一个迭代器 beg：

```
remove_reference<decltype(*beg)>::type
```

将获得 beg 引用的元素的类型：decltype(*beg) 返回元素类型的引用类型。remove_reference::type 脱去引用，剩下元素类型本身。

组合使用 remove_reference、尾置返回及 decltype，我们就可以在函数中返回元素值的拷贝：

685

```
// 为了使用模板参数的成员，必须用 typename，参见 16.1.3 节（第 593 页）
template <typename It>
auto fcn2(It beg, It end) ->
    typename remove_reference<decltype(*beg)>::type
{
    // 处理序列
    return *beg; // 返回序列中一个元素的拷贝
}
```

注意，type 是一个类的成员，而该类依赖于一个模板参数。因此，我们必须在返回类型的声明中使用 typename 来告知编译器，type 表示一个类型（参见 16.1.3 节，第 593 页）

表 16.1：标准类型转换模板

对 Mod<T>，其中 Mod 为	若 T 为	则 Mod<T>::type 为
remove_reference	X&或 X&&	X
	否则	T
add_const	X&、const X 或函数	T
	否则	const T
add_lvalue_reference	X&	T
	X&&	X&
	否则	T&
add_rvalue_reference	X&或 X&&	T
	否则	T&&
remove_pointer	X*	X
	否则	T
add_pointer	X&或 X&&	X*
	否则	T*
make_signed	unsigned X	X
	否则	T
make_unsigned	带符号类型	unsigned X
	否则	T
remove_extent	X[n]	X
	否则	T
remove_all_extents	X[n1][n2]…	X
	否则	T

表 16.1 中描述的每个类型转换模板的工作方式都与 remove_reference 类似。每个模板都有一个名为 type 的 public 成员，表示一个类型。此类型与模板自身的模板类型参数相关，其关系如模板名所示。如果不可能（或者不必要）转换模板参数，则 type 成员就是模板参数类型本身。例如，如果 T 是一个指针类型，则 remove_pointer<T>::type 是 T 指向的类型。如果 T 不是一个指针，则无须进行任何

转换，从而 type 具有与 T 相同的类型。

16.2.3 节练习

686

练习 16.40：下面的函数是否合法？如果不合法，为什么？如果合法，对可以传递的实参类型有什么限制（如果有的话）？返回类型是什么？

```
template <typename It>
auto fcn3(It beg, It end) -> decltype(*beg + 0)
{
    // 处理序列
    return *beg; // 返回序列中一个元素的拷贝
}
```

练习 16.41：编写一个新的 sum 版本，它的返回类型保证足够大，足以容纳加法结果。

16.2.4 函数指针和实参推断

当我们用一个函数模板初始化一个函数指针或为一个函数指针赋值（参见 6.7 节，第 221 页）时，编译器使用指针的类型来推断模板实参。

例如，假定我们有一个函数指针，它指向的函数返回 int，接受两个参数，每个参数都是指向 const int 的引用。我们可以使用该指针指向 compare 的一个实例：

```
template <typename T> int compare(const T&, const T&);
// pf1 指向实例 int compare(const int&, const int&)
int (*pf1)(const int&, const int&) = compare;
```

pf1 中参数的类型决定了 T 的模板实参的类型。在本例中，T 的模板实参类型为 int。指针 pf1 指向 compare 的 int 版本实例。如果不能从函数指针类型确定模板实参，则产生错误：

```
// func 的重载版本；每个版本接受一个不同的函数指针类型
void func(int(*)(const string&, const string&));
void func(int(*)(const int&, const int&));
func(compare); // 错误：使用 compare 的哪个实例？
```

这段代码的问题在于，通过 func 的参数类型无法确定模板实参的唯一类型。对 func 的调用既可以实例化接受 int 的 compare 版本，也可以实例化接受 string 的版本。由于不能确定 func 的实参的唯一实例化版本，此调用将编译失败。

我们可以通过使用显式模板实参来消除 func 调用的歧义：

```
// 正确：显式指出实例化哪个 compare 版本
func(compare<int>); // 传递 compare(const int&, const int&)
```

此表达式调用的 func 版本接受一个函数指针，该指针指向的函数接受两个 const int& 参数。

当参数是一个函数模板实例的地址时，程序上下文必须满足：对每个模板参数，能唯一确定其类型或值。

687

16.2.5 模板实参推断和引用

为了理解如何从函数调用进行类型推断，考虑下面的例子：

```
template <typename T> void f(T &p);
```

其中函数参数 p 是一个模板类型参数 T 的引用，非常重要的是记住两点：编译器会应用正常的引用绑定规则；const 是底层的，不是顶层的。

从左值引用函数参数推断类型

当一个函数参数是模板类型参数的一个普通（左值）引用时（即，形如 T&），绑定规则告诉我们，只能传递给它一个左值（如，一个变量或一个返回引用类型的表达式）。实参可以是 const 类型，也可以不是。如果实参是 const 的，则 T 将被推断为 const 类型：

```
template <typename T> void f1(T&); // 实参必须是一个左值
// 对 f1 的调用使用实参所引用的类型作为模板参数类型
f1(i);   // i 是一个 int；模板参数类型 T 是 int
f1(ci);  // ci 是一个 const int；模板参数 T 是 const int
f1(5);   // 错误：传递给一个&参数的实参必须是一个左值
```

如果一个函数参数的类型是 const T&，正常的绑定规则告诉我们可以传递给它任何类型的实参——一个对象（const 或非 const）、一个临时对象或是一个字面常量值。当函数参数本身是 const 时，T 的类型推断的结果不会是一个 const 类型。const 已经是函数参数类型的一部分；因此，它不会也是模板参数类型的一部分：

```
template <typename T> void f2(const T&); // 可以接受一个右值
// f2 中的参数是 const &；实参中的 const 是无关的
// 在每个调用中，f2 的函数参数都被推断为 const int&
f2(i);   // i 是一个 int；模板参数 T 是 int
f2(ci);  // ci 是一个 const int，但模板参数 T 是 int
f2(5);   // 一个 const &参数可以绑定到一个右值；T 是 int
```

从右值引用函数参数推断类型

当一个函数参数是一个右值引用（参见 13.6.1 节，第 471 页）（即，形如 T&&）时，正常绑定规则告诉我们可以传递给它一个右值。当我们这样做时，类型推断过程类似普通左值引用函数参数的推断过程。推断出的 T 的类型是该右值实参的类型：

```
template <typename T> void f3(T&&);
f3(42); // 实参是一个 int 类型的右值；模板参数 T 是 int
```

688 引用折叠和右值引用参数

假定 i 是一个 int 对象，我们可能认为像 f3(i)这样的调用是不合法的。毕竟，i 是一个左值，而通常我们不能将一个右值引用绑定到一个左值上。但是，C++语言在正常绑定规则之外定义了两个例外规则，允许这种绑定。这两个例外规则是 move 这种标准库设施正确工作的基础。

第一个例外规则影响右值引用参数的推断如何进行。当我们将一个左值（如 i）传递给函数的右值引用参数，且此右值引用指向模板类型参数（如 T&&）时，编译器推断模板类型参数为实参的左值引用类型。因此，当我们调用 f3(i)时，编译器推断 T 的类型为 int&，而非 int。

T 被推断为 int&看起来好像意味着 f3 的函数参数应该是一个类型 int&的右值引用。

通常，我们不能（直接）定义一个引用的引用（参见 2.3.1 节，第 46 页）。但是，通过类型别名（参见 2.5.1 节，第 60 页）或通过模板类型参数间接定义是可以的。

在这种情况下，我们可以使用第二个例外绑定规则：如果我们间接创建一个引用的引用，则这些引用形成了“折叠”。在所有情况下（除了一个例外），引用会折叠成一个普通的左值引用类型。在新标准中，折叠规则扩展到右值引用。只在一种特殊情况下引用会折叠成右值引用：右值引用的右值引用。即，对于一个给定类型 X：

C++ 11

- X& &、X& &&和 X&& &都折叠成类型 X&
- 类型 X&& &&折叠成 X&&

引用折叠只能应用于间接创建的引用的引用，如类型别名或模板参数。

如果将引用折叠规则和右值引用的特殊类型推断规则组合在一起，则意味着我们可以对一个左值调用 f3。当我们将一个左值传递给 f3 的（右值引用）函数参数时，编译器推断 T 为一个左值引用类型：

```
f3(i);   // 实参是一个左值；模板参数 T 是 int&
f3(ci);  // 实参是一个左值；模板参数 T 是一个 const int&
```

当一个模板参数 T 被推断为引用类型时，折叠规则告诉我们函数参数 T&&折叠为一个左值引用类型。例如，f3(i)的实例化结果可能像下面这样：

```
// 无效代码，只是用于演示目的
void f3<int&>(int& &&); // 当 T 是 int&时，函数参数为 int& &&
```

f3 的函数参数是 T&&且 T 是 int&，因此 T&&是 int& &&，会折叠成 int&。因此，即使 f3 的函数参数形式是一个右值引用（即，T&&），此调用也会用一个左值引用类型（即，int&）实例化 f3:

```
void f3<int&>(int&); // 当 T 是 int&时，函数参数折叠为 int&
```

这两个规则导致了两个重要结果：

689

- 如果一个函数参数是一个指向模板类型参数的右值引用（如，T&&），则它可以被绑定到一个左值；且
- 如果实参是一个左值，则推断出的模板实参类型将是一个左值引用，且函数参数将被实例化为一个（普通）左值引用参数（T&）

另外值得注意的是，这两个规则暗示，我们可以将任意类型的实参传递给 T&&类型的函数参数。对于这种类型的参数，（显然）可以传递给它右值，而如我们刚刚看到的，也可以传递给它左值。

如果一个函数参数是指向模板参数类型的右值引用（如，T&&），则可以传递给它任意类型的实参。如果将一个左值传递给这样的参数，则函数参数被实例化为一个普通的左值引用（T&）。

编写接受右值引用参数的模板函数

模板参数可以推断为一个引用类型，这一特性对模板内的代码可能有令人惊讶的影响：

```
template <typename T> void f3(T&& val)
{
    T t = val; // 拷贝还是绑定一个引用？
```

```
    t = fcn(t); // 赋值只改变 t 还是既改变 t 又改变 val?
    if (val == t) { /* ... */ } // 若 T 是引用类型，则一直为 true
}
```

当我们对一个右值调用 f3 时，例如字面常量 42，T 为 int。在此情况下，局部变量 t 的类型为 int，且通过拷贝参数 val 的值被初始化。当我们对 t 赋值时，参数 val 保持不变。

另一方面，当我们对一个左值 i 调用 f3 时，则 T 为 int&。当我们定义并初始化局部变量 t 时，赋予它类型 int&。因此，对 t 的初始化将其绑定到 val。当我们对 t 赋值时，也同时改变了 val 的值。在 f3 的这个实例化版本中，if 判断永远得到 true。

当代码中涉及的类型可能是普通（非引用）类型，也可能是引用类型时，编写正确的代码就变得异常困难（虽然 remove_reference 这样的类型转换类可能会有帮助（参见 16.2.3 节，第 605 页））。

在实际中，右值引用通常用于两种情况：模板转发其实参或模板被重载。我们将在 16.2.7 节（第 612 页）中介绍实参转发，在 16.3 节（第 614 页）中介绍模板重载。

目前应该注意的是，使用右值引用的函数模板通常使用我们在 13.6.3 节（第 481 页）中看到的方式来进行重载：

```
template <typename T> void f(T&&);          // 绑定到非 const 右值
template <typename T> void f(const T&);     // 左值和 const 右值
```

与非模板函数一样，第一个版本将绑定到可修改的右值，而第二个版本将绑定到左值或 const 右值。

690

16.2.5 节练习

练习 16.42：对下面每个调用，确定 T 和 val 的类型：

```
template <typename T> void g(T&& val);
int i = 0; const int ci = i;
(a) g(i); (b) g(ci); (c) g(i * ci);
```

练习 16.43：使用上一题定义的函数，如果我们调用 g(i = ci)，g 的模板参数将是什么？

练习 16.44：使用与第一题中相同的三个调用，如果 g 的函数参数声明为 T（而不是 T&&），确定 T 的类型。如果 g 的函数参数是 const T&呢？

练习 16.45：给定下面的模板，如果我们对一个像 42 这样的字面常量调用 g，解释会发生什么？如果我们对一个 int 类型的变量调用 g 呢？

```
template <typename T> void g(T&& val) { vector<T> v; }
```

16.2.6 理解 std::move

标准库 move 函数（参见 13.6.1 节，第 472 页）是使用右值引用的模板的一个很好的例子。幸运的是，我们不必理解 move 所使用的模板机制也可以直接使用它。但是，研究 move 是如何工作的可以帮助我们巩固对模板的理解和使用。

在 13.6.2 节（第 473 页）中我们注意到，虽然不能直接将一个右值引用绑定到一个左值上，但可以用 move 获得一个绑定到左值上的右值引用。由于 move 本质上可以接受任

何类型的实参，因此我们不会惊讶于它是一个函数模板。

std::move 是如何定义的

标准库是这样定义 move 的：

```
// 在返回类型和类型转换中也要用到 typename，参见 16.1.3 节（第 593 页）
// remove_reference 是在 16.2.3 节（第 605 页）中介绍的
template <typename T>
typename remove_reference<T>::type&& move(T&& t)
{
    // static_cast 是在 4.11.3 节（第 145 页）中介绍的
    return static_cast<typename remove_reference<T>::type&&>(t);
}
```

这段代码很短，但其中有些微妙之处。首先，move 的函数参数 T&&是一个指向模板类型参数的右值引用。通过引用折叠，此参数可以与任何类型的实参匹配。特别是，我们既可以传递给 move 一个左值，也可以传递给它一个右值：

```
string s1("hi!"), s2;
s2 = std::move(string("bye!")); // 正确：从一个右值移动数据
s2 = std::move(s1); // 正确：但在赋值之后，s1 的值是不确定的
```

std::move 是如何工作的

在第一个赋值中，传递给 move 的实参是 string 的构造函数的右值结果——string("bye!")。如我们已经见到过的，当向一个右值引用函数参数传递一个右值时，由实参推断出的类型为被引用的类型（参见 16.2.5 节，第 608 页）。因此，在 std::move(string("bye!"))中：

691

- 推断出的 T 的类型为 string。
- 因此，remove_reference 用 string 进行实例化。
- remove_reference<string>的 type 成员是 string。
- move 的返回类型是 string&&。
- move 的函数参数 t 的类型为 string&&。

因此，这个调用实例化 move<string>，即函数

```
string&& move(string &&t)
```

函数体返回 static_cast<string&&>(t)。t 的类型已经是 string&&，于是类型转换什么都不做。因此，此调用的结果就是它所接受的右值引用。

现在考虑第二个赋值，它调用了 std::move()。在此调用中，传递给 move 的实参是一个左值。这样：

- 推断出的 T 的类型为 string&（string 的引用，而非普通 string）。
- 因此，remove_reference 用 string&进行实例化。
- remove_reference<string&>的 type 成员是 string。
- move 的返回类型仍是 string&&。
- move 的函数参数 t 实例化为 string& &&，会折叠为 string&。

因此，这个调用实例化 move<string&>，即

```
string&& move(string &t)
```

这正是我们所寻求的——我们希望将一个右值引用绑定到一个左值。这个实例的函数体返回 static_cast<string&&>(t)。在此情况下，t 的类型为 string&，cast 将其转换为 string&&。

从一个左值 static_cast 到一个右值引用是允许的

C++ 11

通常情况下，static_cast 只能用于其他合法的类型转换（参见 4.11.3 节，第 145 页）。但是，这里又有一条针对右值引用的特许规则：虽然不能隐式地将一个左值转换为右值引用，但我们可以用 static_cast 显式地将一个左值转换为一个右值引用。

692

对于操作右值引用的代码来说，将一个右值引用绑定到一个左值的特性允许它们截断左值。有时候，例如在我们的 StrVec 类的 reallocate 函数（参见 13.6.1 节，第 469 页）中，我们知道截断一个左值是安全的。一方面，通过允许进行这样的转换，C++语言认可了这种用法。但另一方面，通过强制使用 static_cast，C++语言试图阻止我们意外地进行这种转换。

最后，虽然我们可以直接编写这种类型转换代码，但使用标准库 move 函数是容易得多的方式。而且，统一使用 std::move 使得我们在程序中查找潜在的截断左值的代码变得很容易。

16.2.6 节练习

练习 16.46：解释下面的循环，它来自 13.5 节（第 469 页）中的 StrVec::reallocate：

```
for (size_t i = 0; i != size(); ++i)
    alloc.construct(dest++, std::move(*elem++));
```

16.2.7 转发

某些函数需要将其一个或多个实参连同类型不变地转发给其他函数。在此情况下，我们需要保持被转发实参的所有性质，包括实参类型是否是 const 的以及实参是左值还是右值。

作为一个例子，我们将编写一个函数，它接受一个可调用表达式和两个额外实参。我们的函数将调用给定的可调用对象，将两个额外参数逆序传递给它。下面是我们的翻转函数的初步模样：

```
// 接受一个可调用对象和另外两个参数的模板
// 对“翻转”的参数调用给定的可调用对象
// flip1 是一个不完整的实现：顶层 const 和引用丢失了
template <typename F, typename T1, typename T2>
void flip1(F f, T1 t1, T2 t2)
{
    f(t2, t1);
}
```

这个函数一般情况下工作得很好，但当我们希望用它调用一个接受引用参数的函数时就会出现问题：

```
void f(int v1, int &v2) // 注意 v2 是一个引用
{
    cout << v1 << " " << ++v2 << endl;
}
```

在这段代码中，f 改变了绑定到 v2 的实参的值。但是，如果我们通过 flip1 调用 f，f 所做的改变就不会影响实参：

```
f(42, i);          // f 改变了实参 i
flip1(f, j, 42); // 通过 flip1 调用 f 不会改变 j
```

问题在于 j 被传递给 flip1 的参数 t1。此参数是一个普通的、非引用的类型 int，而非 int&。因此，这个 flip1 调用会实例化为

```
void flip1(void(*fcn)(int, int&), int t1, int t2);
```

693

j 的值被拷贝到 t1 中。f 中的引用参数被绑定到 t1，而非 j，从而其改变不会影响 j。

定义能保持类型信息的函数参数

为了通过翻转函数传递一个引用，我们需要重写函数，使其参数能保持给定实参的“左值性”。更进一步，可以想到我们也希望保持参数的 const 属性。

通过将一个函数参数定义为一个指向模板类型参数的右值引用，我们可以保持其对应实参的所有类型信息。而使用引用参数（无论是左值还是右值）使得我们可以保持 const 属性，因为在引用类型中的 const 是底层的。如果我们将函数参数定义为 T1&&和 T2&&，通过引用折叠（参见 16.2.5 节，第 608 页）就可以保持翻转实参的左值/右值属性（参见 16.2.5 节，第 608 页）：

```
template <typename F, typename T1, typename T2>
void flip2(F f, T1 &&t1, T2 &&t2)
{
    f(t2, t1);
}
```

与较早的版本一样，如果我们调用 flip2(f, j, 42)，将传递给参数 t1 一个左值 j。但是，在 flip2 中，推断出的 T1 的类型为 int&，这意味着 t1 的类型会折叠为 int&。由于是引用类型，t1 被绑定到 j 上。当 flip2 调用 f 时，f 中的引用参数 v2 被绑定到 t1，也就是被绑定到 j。当 f 递增 v2 时，它也同时改变了 j 的值。

如果一个函数参数是指向模板类型参数的右值引用（如 T&&），它对应的实参的 const 属性和左值/右值属性将得到保持。

这个版本的 flip2 解决了一半问题。它对于接受一个左值引用的函数工作得很好，但不能用于接受右值引用参数的函数。例如：

```
void g(int &&i, int& j)
{
    cout << i << " " << j << endl;
}
```

如果我们试图通过 flip2 调用 g，则参数 t2 将被传递给 g 的右值引用参数。即使我们传递一个右值给 flip2：

```
flip2(g, i, 42); // 错误：不能从一个左值实例化 int&&
```

传递给 g 的将是 flip2 中名为 t2 的参数。函数参数与其他任何变量一样，都是左值表达式（参见 13.6.1 节，第 471 页）。因此，flip2 中对 g 的调用将传递给 g 的右值引用参数一个左值。

在调用中使用 std::forward 保持类型信息

694 我们可以使用一个名为 forward 的新标准库设施来传递 flip2 的参数，它能保持原始实参的类型。类似 move，forward 定义在头文件 utility 中。与 move 不同，forward 必须通过显式模板实参来调用（参见 16.2.2 节，第 603 页）。forward 返回该显式实参类型的右值引用。即，forward<T>的返回类型是 T&&。

C++ 11

通常情况下，我们使用 forward 传递那些定义为模板类型参数的右值引用的函数参数。通过其返回类型上的引用折叠，forward 可以保持给定实参的左值/右值属性：

```
template <typename Type> intermediary(Type &&arg)
{
    finalFcn(std::forward<Type>(arg));
    // ...
}
```

本例中我们使用 Type 作为 forward 的显式模板实参类型，它是从 arg 推断出来的。由于 arg 是一个模板类型参数的右值引用，Type 将表示传递给 arg 的实参的所有类型信息。如果实参是一个右值，则 Type 是一个普通（非引用）类型，forward<Type>将返回 Type&&。如果实参是一个左值，则通过引用折叠，Type 本身是一个左值引用类型。在此情况下，返回类型是一个指向左值引用类型的右值引用。再次对 forward<Type>的返回类型进行引用折叠，将返回一个左值引用类型。

当用于一个指向模板参数类型的右值引用函数参数（T&&）时，forward 会保持实参类型的所有细节。

使用 forward，我们可以再次重写翻转函数：

```
template <typename F, typename T1, typename T2>
void flip(F f, T1 &&t1, T2 &&t2)
{
    f(std::forward<T2>(t2), std::forward<T1>(t1));
}
```

如果我们调用 flip(g, i, 42)，i 将以 int&类型传递给 g，42 将以 int&&类型传递给 g。

与 std::move 相同，对 std::forward 不使用 using 声明是一个好主意。我们将在 18.2.3 节（第 706 页）中解释原因。

16.2.7 节练习

练习 16.47：编写你自己版本的翻转函数，通过调用接受左值和右值引用参数的函数来测试它。

16.3 重载与模板

函数模板可以被另一个模板或一个普通非模板函数重载。与往常一样，名字相同的函数必须具有不同数量或类型的参数。

695 如果涉及函数模板，则函数匹配规则（参见 6.4 节，第 209 页）会在以下几方面受到

影响：

- 对于一个调用，其候选函数包括所有模板实参推断（参见 16.2 节，第 600 页）成功的函数模板实例。
- 候选的函数模板总是可行的，因为模板实参推断会排除任何不可行的模板。
- 与往常一样，可行函数（模板与非模板）按类型转换（如果对此调用需要的话）来排序。当然，可以用于函数模板调用的类型转换是非常有限的（参见 16.2.1 节，第 601 页）。
- 与往常一样，如果恰有一个函数提供比任何其他函数都更好的匹配，则选择此函数。但是，如果有多个函数提供同样好的匹配，则：
  — 如果同样好的函数中只有一个是非模板函数，则选择此函数。
  — 如果同样好的函数中没有非模板函数，而有多个函数模板，且其中一个模板比其他模板更特例化，则选择此模板。
  — 否则，此调用有歧义。

WARNING

正确定义一组重载的函数模板需要对类型间的关系及模板函数允许的有限的实参类型转换有深刻的理解。

编写重载模板

作为一个例子，我们将构造一组函数，它们在调试中可能很有用。我们将这些调试函数命名为 debug_rep，每个函数都返回一个给定对象的 string 表示。我们首先编写此函数的最通用版本，将它定义为一个模板，接受一个 const 对象的引用：

```
// 打印任何我们不能处理的类型
template <typename T> string debug_rep(const T &t)
{
    ostringstream ret; // 参见 8.3 节（第 287 页）
    ret << t; // 使用 T 的输出运算符打印 t 的一个表示形式
    return ret.str(); // 返回 ret 绑定的 string 的一个副本
}
```

此函数可以用来生成一个对象对应的 string 表示，该对象可以是任意具备输出运算符的类型。 696

接下来，我们将定义打印指针的 debug_rep 版本：

```
// 打印指针的值，后跟指针指向的对象
// 注意：此函数不能用于 char*；参见 16.3 节（第 617 页）
template <typename T> string debug_rep(T *p)
{
    ostringstream ret;
    ret << "pointer: " << p;      // 打印指针本身的值
    if (p)
        ret << " " << debug_rep(*p); // 打印 p 指向的值
    else
        ret << " null pointer";  // 或指出 p 为空
    return ret.str(); // 返回 ret 绑定的 string 的一个副本
}
```

此版本生成一个 string，包含指针本身的值和调用 debug_rep 获得的指针指向的值。注意此函数不能用于打印字符指针，因为 IO 库为 char*值定义了一个<<版本。此<<版本假定指针表示一个空字符结尾的字符数组，并打印数组的内容而非地址值。我们将在 16.3

节（第 617 页）介绍如何处理字符指针。

我们可以这样使用这些函数：

```
string s("hi");
cout << debug_rep(s) << endl;
```

对于这个调用，只有第一个版本的 debug_rep 是可行的。第二个 debug_rep 版本要求一个指针参数，但在此调用中我们传递的是一个非指针对象。因此编译器无法从一个非指针实参实例化一个期望指针类型参数的函数模板，因此实参推断失败。由于只有一个可行函数，所以此函数被调用。

如果我们用一个指针调用 debug_rep：

```
cout << debug_rep(&s) << endl;
```

两个函数都生成可行的实例：

- debug_rep(const string*&)，由第一个版本的 debug_rep 实例化而来，T 被绑定到 string*。
- debug_rep(string*)，由第二个版本的 debug_rep 实例化而来，T 被绑定到 string。

第二个版本的 debug_rep 的实例是此调用的精确匹配。第一个版本的实例需要进行普通指针到 const 指针的转换。正常函数匹配规则告诉我们应该选择第二个模板，实际上编译器确实选择了这个版本。

697

多个可行模板

作为另外一个例子，考虑下面的调用：

```
const string *sp = &s;
cout << debug_rep(sp) << endl;
```

此例中的两个模板都是可行的，而且两个都是精确匹配：

- debug_rep(const string*&)，由第一个版本的 debug_rep 实例化而来，T 被绑定到 string*。
- debug_rep(const string*)，由第二个版本的 debug_rep 实例化而来，T 被绑定到 const string。

在此情况下，正常函数匹配规则无法区分这两个函数。我们可能觉得这个调用将是有歧义的。但是，根据重载函数模板的特殊规则，此调用被解析为 debug_rep(T*)，即，更特例化的版本。

设计这条规则的原因是，没有它，将无法对一个 const 的指针调用指针版本的 debug_rep。问题在于模板 debug_rep(const T&) 本质上可以用于任何类型，包括指针类型。此模板比 debug_rep(T*) 更通用，后者只能用于指针类型。没有这条规则，传递 const 的指针的调用永远是有歧义的。

当有多个重载模板对一个调用提供同样好的匹配时，应选择最特例化的版本。

非模板和模板重载

作为下一个例子，我们将定义一个普通非模板版本的 debug_rep 来打印双引号包围

的 string：

```
// 打印双引号包围的 string
string debug_rep(const string &s)
{
    return '"' + s + '"';
}
```

现在，当我们对一个 string 调用 debug_rep 时：

```
string s("hi");
cout << debug_rep(s) << endl;
```

有两个同样好的可行函数：

- debug_rep<string>(const string&)，第一个模板，T 被绑定到 string*。
- debug_rep(const string&)，普通非模板函数。

在本例中，两个函数具有相同的参数列表，因此显然两者提供同样好的匹配。但是，编译器会选择非模板版本。当存在多个同样好的函数模板时，编译器选择最特例化的版本，出于相同的原因，一个非模板函数比一个函数模板更好。 698

对于一个调用，如果一个非函数模板与一个函数模板提供同样好的匹配，则选择非模板版本。

重载模板和类型转换

还有一种情况我们到目前为止尚未讨论：C 风格字符串指针和字符串字面常量。现在有了一个接受 string 的 debug_rep 版本，我们可能期望一个传递字符串的调用会匹配这个版本。但是，考虑这个调用：

```
cout << debug_rep("hi world!") << endl; // 调用 debug_rep(T*)
```

本例中所有三个 debug_rep 版本都是可行的：

- debug_rep(const T&)，T 被绑定到 char[10]。
- debug_rep(T*)，T 被绑定到 const char。
- debug_rep(const string&)，要求从 const char*到 string 的类型转换。

对给定实参来说，两个模板都提供精确匹配——第二个模板需要进行一次（许可的）数组到指针的转换，而对于函数匹配来说，这种转换被认为是精确匹配（参见 6.6.1 节，第 219 页）。非模板版本是可行的，但需要进行一次用户定义的类型转换，因此它没有精确匹配那么好，所以两个模板成为可能调用的函数。与之前一样，T*版本更加特例化，编译器会选择它。

如果我们希望将字符指针按 string 处理，可以定义另外两个非模板重载版本：

```
// 将字符指针转换为 string，并调用 string 版本的 debug_reg
string debug_rep(char *p)
{
    return debug_rep(string(p));
}
string debug_rep(const char *p)
{
    return debug_rep(string(p));
}
```

缺少声明可能导致程序行为异常

值得注意的是，为了使 char*版本的 debug_rep 正确工作，在定义此版本时，debug_rep(const string&)的声明必须在作用域中。否则，就可能调用错误的 debug_rep 版本：

699

```
template <typename T> string debug_rep(const T &t);
template <typename T> string debug_rep(T *p);
// 为了使 debug_rep(char*)的定义正确工作，下面的声明必须在作用域中
string debug_rep(const string &);
string debug_rep(char *p)
{
    // 如果接受一个 const string&的版本的声明不在作用域中，
    // 返回语句将调用 debug_rep(const T&)的 T 实例化为 string 的版本
    return debug_rep(string(p));
}
```

通常，如果使用了一个忘记声明的函数，代码将编译失败。但对于重载函数模板的函数而言，则不是这样。如果编译器可以从模板实例化出与调用匹配的版本，则缺少的声明就不重要了。在本例中，如果忘记了声明接受 string 参数的 debug_rep 版本，编译器会默默地实例化接受 const T&的模板版本。

在定义任何函数之前，记得声明所有重载的函数版本。这样就不必担心编译器由于未遇到你希望调用的函数而实例化一个并非你所需的版本。

16.3 节练习

练习 16.48：编写你自己版本的 debug_rep 函数。

练习 16.49：解释下面每个调用会发生什么：

```
template <typename T> void f(T);
template <typename T> void f(const T*);
template <typename T> void g(T);
template <typename T> void g(T*);
int i = 42, *p = &i;
const int ci = 0, *p2 = &ci;
g(42); g(p); g(ci); g(p2);
f(42); f(p); f(ci); f(p2);
```

练习 16.50：定义上一个练习中的函数，令它们打印一条身份信息。运行该练习中的代码。如果函数调用的行为与你预期不符，确定你理解了原因。

16.4 可变参数模板

C++ 11

一个**可变参数模板**（variadic template）就是一个接受可变数目参数的模板函数或模板类。可变数目的参数被称为**参数包**（parameter packet）。存在两种参数包：**模板参数包**（template parameter packet），表示零个或多个模板参数；**函数参数包**（function parameter packet），表示零个或多个函数参数。

700

我们用一个省略号来指出一个模板参数或函数参数表示一个包。在一个模板参数列表

中，class...或 typename...指出接下来的参数表示零个或多个类型的列表；一个类型名后面跟一个省略号表示零个或多个给定类型的非类型参数的列表。在函数参数列表中，如果一个参数的类型是一个模板参数包，则此参数也是一个函数参数包。例如：

```
// Args 是一个模板参数包；rest 是一个函数参数包
// Args 表示零个或多个模板类型参数
// rest 表示零个或多个函数参数
template <typename T, typename... Args>
void foo(const T &t, const Args& ... rest);
```

声明了 foo 是一个可变参数函数模板，它有一个名为 T 的类型参数，和一个名为 Args 的模板参数包。这个包表示零个或多个额外的类型参数。foo 的函数参数列表包含一个 const &类型的参数，指向 T 的类型，还包含一个名为 rest 的函数参数包，此包表示零个或多个函数参数。

与往常一样，编译器从函数的实参推断模板参数类型。对于一个可变参数模板，编译器还会推断包中参数的数目。例如，给定下面的调用：

```
int i = 0; double d = 3.14; string s = "how now brown cow";
foo(i, s, 42, d);     // 包中有三个参数
foo(s, 42, "hi");     // 包中有两个参数
foo(d, s);            // 包中有一个参数
foo("hi");            // 空包
```

编译器会为 foo 实例化出四个不同的版本：

```
void foo(const int&, const string&, const int&, const double&);
void foo(const string&, const int&, const char[3]&);
void foo(const double&, const string&);
void foo(const char[3]&);
```

在每个实例中，T 的类型都是从第一个实参的类型推断出来的。剩下的实参（如果有的话）提供函数额外实参的数目和类型。

sizeof...运算符

当我们需要知道包中有多少元素时，可以使用 sizeof...运算符。类似 sizeof（参见 4.9 节，第 139 页），sizeof...也返回一个常量表达式（参见 2.4.4 节，第 58 页），而且不会对其实参求值：

C++ 11

```
template<typename ... Args> void g(Args ... args) {
    cout << sizeof...(Args) << endl; // 类型参数的数目
    cout << sizeof...(args) << endl; // 函数参数的数目
}
```

16.4 节练习

701

练习 16.51：调用本节中的每个 foo，确定 sizeof...(Args)和 sizeof...(rest)分别返回什么。

练习 16.52：编写一个程序验证上一题的答案。

16.4.1 编写可变参数函数模板

如 6.2.6 节（第 198 页）所述，我们可以使用一个 initializer_list 来定义一个可接受可变数目实参的函数。但是，所有实参必须具有相同的类型（或它们的类型可以转换为同一个公共类型）。当我们既不知道想要处理的实参的数目也不知道它们的类型时，可变参数函数是很有用的。作为一个例子，我们将定义一个函数，它类似较早的 error_msg 函数，差别仅在于新函数实参的类型也是可变的。我们首先定义一个名为 print 的函数，它在一个给定流上打印给定实参列表的内容。

可变参数函数通常是递归的（参见 6.3.2 节，第 204 页）。第一步调用处理包中的第一个实参，然后用剩余实参调用自身。我们的 print 函数也是这样的模式，每次递归调用将第二个实参打印到第一个实参表示的流中。为了终止递归，我们还需要定义一个非可变参数的 print 函数，它接受一个流和一个对象：

```
// 用来终止递归并打印最后一个元素的函数
// 此函数必须在可变参数版本的 print 定义之前声明
template<typename T>
ostream &print(ostream &os, const T &t)
{
    return os << t; // 包中最后一个元素之后不打印分隔符
}
// 包中除了最后一个元素之外的其他元素都会调用这个版本的 print
template <typename T, typename... Args>
ostream &print(ostream &os, const T &t, const Args&... rest)
{
    os << t << ", ";              // 打印第一个实参
    return print(os, rest...);    // 递归调用，打印其他实参
}
```

第一个版本的 print 负责终止递归并打印初始调用中的最后一个实参。第二个版本的 print 是可变参数版本，它打印绑定到 t 的实参，并调用自身来打印函数参数包中的剩余值。

这段程序的关键部分是可变参数函数中对 print 的调用：

```
return print(os, rest...); //递归调用，打印其他实参
```

我们的可变参数版本的 print 函数接受三个参数：一个 ostream&，一个 const T&和一个参数包。而此调用只传递了两个实参。其结果是 rest 中的第一个实参被绑定到 t，剩余实参形成下一个 print 调用的参数包。因此，在每个调用中，包中的第一个实参被移除，成为绑定到 t 的实参。即，给定：

702

```
print(cout, i, s, 42); // 包中有两个参数
```

递归会执行如下：

调用	t	rest...
print(cout, i, s, 42)	i	s, 42
print(cout, s, 42)	s	42
print(cout, 42) 调用非可变参数版本的 print		

前两个调用只能与可变参数版本的 print 匹配，非可变参数版本是不可行的，因为这两个调用分别传递四个和三个实参，而非可变参数 print 只接受两个实参。

对于最后一次递归调用 print(cout, 42)，两个 print 版本都是可行的。这个调用传递两个实参，第一个实参的类型为 ostream&。因此，可变参数版本的 print 可以实例化为只接受两个参数：一个是 ostream&参数，另一个是 const T&参数。

对于最后一个调用，两个函数提供同样好的匹配。但是，非可变参数模板比可变参数模板更特例化，因此编译器选择非可变参数版本（参见 16.3 节，第 615 页）。

WARNING

当定义可变参数版本的 print 时，非可变参数版本的声明必须在作用域中。否则，可变参数版本会无限递归。

16.4.1 节练习

练习 16.53：编写你自己版本的 print 函数，并打印一个、两个及五个实参来测试它，要打印的每个实参都应有不同的类型。

练习 16.54：如果我们对一个没有<<运算符的类型调用 print，会发生什么？

练习 16.55：如果我们的可变参数版本 print 的定义之后声明非可变参数版本，解释可变参数的版本会如何执行。

16.4.2 包扩展

对于一个参数包，除了获取其大小外，我们能对它做的唯一的事情就是**扩展**（expand）它。当扩展一个包时，我们还要提供用于每个扩展元素的**模式**（pattern）。扩展一个包就是将它分解为构成的元素，对每个元素应用模式，获得扩展后的列表。我们通过在模式右边放一个省略号（...）来触发扩展操作。 703

例如，我们的 print 函数包含两个扩展：

```
template <typename T, typename... Args>
ostream &
print(ostream &os, const T &t, const Args&... rest)    // 扩展 Args
{
    os << t << ", ";
    return print(os, rest...);                         // 扩展 rest
}
```

第一个扩展操作扩展模板参数包，为 print 生成函数参数列表。第二个扩展操作出现在对 print 的调用中。此模式为 print 调用生成实参列表。

对 Args 的扩展中，编译器将模式 const Arg&应用到模板参数包 Args 中的每个元素。因此，此模式的扩展结果是一个逗号分隔的零个或多个类型的列表，每个类型都形如 const type&。例如：

```
print(cout, i, s, 42); // 包中有两个参数
```

最后两个实参的类型和模式一起确定了尾置参数的类型。此调用被实例化为：

```
ostream&
print(ostream&, const int&, const string&, const int&);
```

第二个扩展发生在对 print 的（递归）调用中。在此情况下，模式是函数参数包的名字（即 rest）。此模式扩展出一个由包中元素组成的、逗号分隔的列表。因此，这个调

用等价于：

```
print(os, s, 42);
```

理解包扩展

print 中的函数参数包扩展仅仅将包扩展为其构成元素，C++语言还允许更复杂的扩展模式。例如，我们可以编写第二个可变参数函数，对其每个实参调用 debug_rep（参见 16.3 节，第 615 页），然后调用 print 打印结果 string：

```
// 在 print 调用中对每个实参调用 debug_rep
template <typename... Args>
ostream &errorMsg(ostream &os, const Args&... rest)
{
    // print(os, debug_rep(a1), debug_rep(a2), ..., debug_rep(an)
    return print(os, debug_rep(rest)...);
}
```

704 这个 print 调用使用了模式 debug_reg(rest)。此模式表示我们希望对函数参数包 rest 中的每个元素调用 debug_rep。扩展结果将是一个逗号分隔的 debug_rep 调用列表。即，下面调用：

```
errorMsg(cerr, fcnName, code.num(), otherData, "other", item);
```

就好像我们这样编写代码一样

```
print(cerr, debug_rep(fcnName), debug_rep(code.num()),
            debug_rep(otherData), debug_rep("otherData"),
            debug_rep(item));
```

与之相对，下面的模式会编译失败

```
// 将包传递给 debug_rep; print(os, debug_rep(a1, a2, ..., an))
print(os, debug_rep(rest...)); // 错误：此调用无匹配函数
```

这段代码的问题是我们在 debug_rep 调用中扩展了 rest，它等价于

```
print(cerr, debug_rep(fcnName, code.num(),
                      otherData, "otherData", item));
```

在这个扩展中，我们试图用一个五个实参的列表来调用 debug_rep，但并不存在与此调用匹配的 debug_rep 版本。debug_rep 函数不是可变参数的，而且没有哪个 debug_rep 版本接受五个参数。

扩展中的模式会独立地应用于包中的每个元素。

16.4.2 节练习

练习 16.56：编写并测试可变参数版本的 errorMsg。

练习 16.57：比较你的可变参数版本的 errorMsg 和 6.2.6 节（第 198 页）中的 error_msg 函数。两种方法的优点和缺点各是什么？

16.4.3 转发参数包

C++ 11 在新标准下，我们可以组合使用可变参数模板与 forward 机制来编写函数，实现将