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	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
unique_ptr<T, D> u2	空 unique_ptr，可以指向类型为 T 的对象。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)	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
u.reset(nullptr)	

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[]。

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！
```

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。

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
487 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。

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。

done!

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。

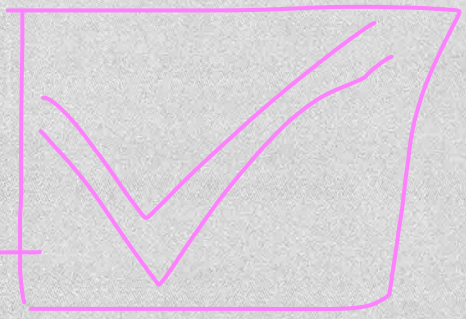
shared_ptr 提供所有权共享的智能指针：对共享对象来说，当最后一个指向它的 shared_ptr 被销毁时会被释放。

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

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。

done!

done!

done!

第 13 章
拷贝控制

内容

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