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

Best Practices 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。

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）。

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
516 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
518
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。

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
520 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。

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。

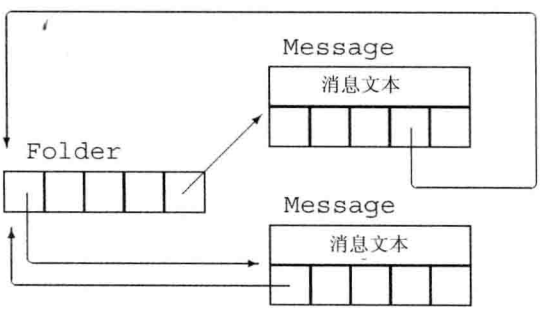


图 13.1：Message 和 Folder 类设计

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
}
```

522

```
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
}
```

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。

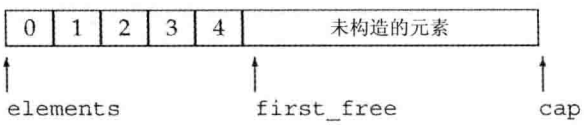


图 13.2：StrVec 内存分配策略

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，
527 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：

```
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。

后置递增

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

C++ 11

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。

```
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

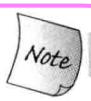
由于移动操作“窃取”资源，它通常不分配任何资源。因此，移动操作通常不会抛出任何异常。当编写一个不抛出异常的移动操作时，我们应该将此事通知标准库。我们将看到，除非标准库知道我们的移动构造函数不会抛出异常，否则它会认为移动我们的类对象时可能会抛出异常，并且为了处理这种可能性而做一些额外的工作。

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
541 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）。

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
546 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。

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：

```
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
```

547

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
    };
```