1，拷贝，复制与销毁

1.1 拷贝构造函数

拷贝构造函数，拷贝赋值运算符，移动构造函数，移动赋值运算符和析构函数。

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。

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。拷贝构造函数不应该是explicit的。

对某些类来说，合成拷贝构造函数用来阻止我们拷贝该类类型的对象。

如果没有定义拷贝构造函数，编译器会为我们合成一个拷贝构造函数。合成拷贝构造函数一般会把参数成员逐个拷贝到正在创建的对象中。

string dots(10, '.'); string s(dots)//直接初始化 string s2 = dots;string null\_book="9-999-99"//拷贝初始化

直接初始化，使用的是构造函数；拷贝初始化使用拷贝构造函数。

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完成。

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。

拷贝初始化不仅在我们用=定义变量时会发生，下列情况也会发生：

1，将一个对象作为实参传递给一个非引用类型的形参；2，从一个返回类型为非引用类型的函数返回一个对象；

3，用花括号列表初始化一个数组中的元素或一个聚合类的成员；4，标准库容器调用insert或push成员

如果我们使用的初始化值要通过一个explicit的构造函数来进行类型转换，那么使用拷贝初始化还是直接初始化就不是无关紧要的。

1.2 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，编译器会为它生成一个。

重载运算符本质上是函数，赋值运算符就是一个名为operator=的函数。

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。

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示参数传递。

赋值运算符通常返回一个指向其左侧运算对象的引用。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。

例子 Sales\_data& Sales\_data::operator=(const Sales\_data &rhs){}

1.3 析构函数

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。

析构函数有一个函数体和一个析构部分。

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。

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。

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，智能指针是类类型，所以具有析构函数。因此，智能指针成员在析构阶段会被自动销毁。

无论何时一个对象被销毁，就会自动调用其析构函数。

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。

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。默认的析构函数是空。

在空析构函数体执行完毕后，成员会被自动销毁。析构函数体自身并不直接销毁成员。成员在析构函数体之后隐含的析构阶段中被销毁。

1.4 三/五法则

有三个基本操作可以控制类的拷贝操作：拷贝析构函数，拷贝赋值运算符和析构函数。

在新的标准下，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。

需要析构函数的类也需要拷贝和赋值操作。

需要拷贝操作的类也需要赋值操作，反之亦然。

1.5 使用=default

我们可以通过将拷贝控制成员定义为=default来显示地要求编译器生成合成版本。例子：见书本（449）

1.6 阻止拷贝

在新标准中，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来阻止拷贝。

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。

NoCopy(const NoCopy&) = delete;NoCopy &operator=(const NoCopy&) =delete;

与default不同，=delete必须出现在函数第一次声明的时候。

析构函数不能是删除的成员。

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。

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

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

声明但不定义一个成员函数是合法的，视图访问一个未定义的成员将导致一个链接时的错误。

通过声明但不定义porivate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。

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。因为需要析构函数来释放对象所分配的资源。

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。

编程：行为像值的类

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先拷贝右侧运算对象。

编程：行为像指针的类

令一个类展示类似指针行为的最好方式是使用shared\_ptr来管理类中的资源。

但是，有时我们希望能直接管理资源。在这种情况下，使用引用计数就很有用了。

在哪里存放引用计数，解决此问题的一种方法是将计算器保存在动态内存中。（用一个指针存放）

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。

对于那些与重排元素顺序的算法一起使用的类，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。

编程：编写我们自己的swap函数。

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

using std::swap;swap(lhs.h,rhs.h)//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，则会使用std中的版本。

不能直接使用std::swap(lhs.h,rhs.h)

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。

编程：在赋值运算符中使用swap。

这个版本的赋值运算符中，参数并不是一个引用，我们将右侧的对象以传值的方式传递给了赋值运算符。

4，拷贝控制示例

编程：Floder和Message类设计（重大实践）

5，动态内存管理类

编程：设计StrVec类（重大实践）

我们将使用一个allocator来获得原始内存。

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。

当我们删除一个元素时，使用destroy成员来销毁元素。

6，对象移动（需要重看）

6.1，右值引用

在某些情况下，对象拷贝后立即被销毁了。在这些情况下，移动而非拷贝对象会大幅度提升性能。

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。

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。

我们通过&&而不是&来获得右值引用，右值引用有一个重要的性质--只能绑定到一个将要摧毁的对象。

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。

右值引用可以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。

int i = 42;int &&rr2 = i \* 42;

左值持久，右值短暂。

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。

变量是左值。

int &&rr1 = 42;//正确 int &&rr2 = rr1;//错误：表达式rr1是左值。

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。

虽然不能将一个右值引用直接绑定到一个左值上，但我们可以显示地将一个左值转换为对应的右值引用类型。

我们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。

int &&rr3 = std::move(rr1);

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。

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

对move我们不提供using声明，我们直接调用std::move而不是move。

6.2 移动构造函数和移动赋值运算符

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，但它们从给定的对象“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。

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。

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。

StrVec::StrVec(StrVec &&s) noexcept:elements(s.elements),first\_free(s.first\_free),cap(s.cap){

s.elements = s.first\_free = s.cap = nullptr;

}

我们必须在类头文件的声明中和定义中都指定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。

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。

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。

编程：检测自赋值的移动运算。

移后源对象必须可析构。

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，但用户不能对其值进行任何假设。

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。

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。

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。

编程：使用合成移动操作。

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，如果我们显示地要求编译器生成=default的移动操作，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。

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，但如果没有移动构造函数，右值也被拷贝。

StrVec v1,v2; v1 = v2;//使用拷贝赋值

StrVec getVec(istream &); v2 = getVec(cin);//使用移动赋值

6.3 右值引用和成员函数