类的拷贝控制成员包括 5 种成员函数：

1.拷贝构造函数：定义了当用同类型的对象初始化另一个对象时做什么

2.拷贝赋值运算符：定义了当将同类型的一个对象赋予另一个对象时做什么

3.移动构造函数：同拷贝构造函数

4.移动赋值运算符：同拷贝赋值运算符

5.析构函数：定义对象销毁时做什么

拷贝控制成员是类的必要部分，如果没有显式定义，编译器会自动为其隐式地定义。

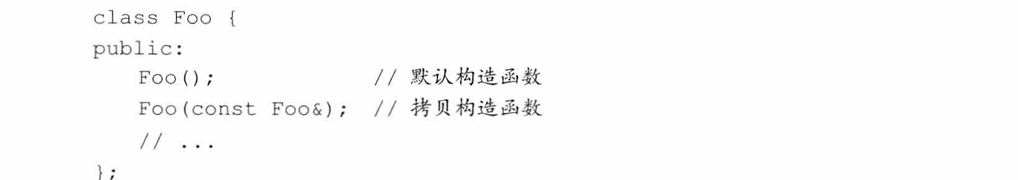
拷贝构造函数特点

1.如果一个构造函数的第一个参数是自身类类型的引用，且其他参数都有默认值，则此构造函数为拷贝构造函数。

2.拷贝构造函数的第一个参数必须是引用，且一般是 const 引用。（如果不使用引用会导致无限循环，因为传递实参本身就是拷贝）

3.拷贝构造函数不能为explicit的，因为在一些情况下会被隐式调用。

书上的例子：



ps：成员类型决定拷贝方式，若为内置类型直接拷贝，为类类型则使用拷贝构造函数拷贝。

合成拷贝构造函数

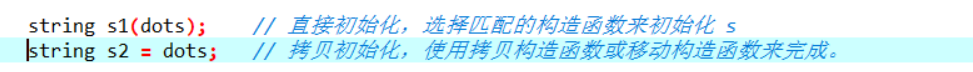
如果没有为类显式定义拷贝构造函数，则编译器会定义一个合成拷贝构造函数。与合成构造函数不同，无论是否有其他构造函数，都会定义一个默认拷贝构造函数，除非已经定义过拷贝构造函数。

一般合成拷贝构造函数会逐个拷贝类的每个成员。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。

对于某些类，合成拷贝构造函数用来禁止该类型对象的拷贝（通过 =delete）

拷贝初始化：

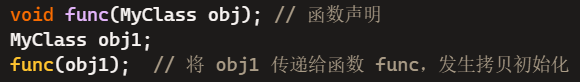
看得出来，用 = 符号即可，编译器从给定对象逐个将每个非static成员拷贝到正在创建的对象中。



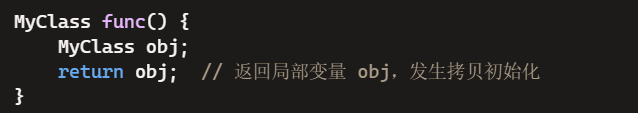
拷贝初始化发生的场合：

1.以 = 定义变量

2.将一个对象作为实参传递给非引用形参



3.从一个返回类型为非引用类型的函数返回一个对象



4.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

5.某些类类型会为它们所分配的对象进行拷贝初始化。比如标准库容器初始化或调用 insert 和 push 成员时，会对其元素进行拷贝初始化（emplace 则是直接初始化）。

C++运算符重载语法：

image.png

return\_type：当前运算符重载函数的返回值

operator：一种特殊的标识符，用于定义或重载运算符

op：被重载的运算符，例如 = 、+等

parameter\_list：运算符的参数列表。

合成拷贝赋值运算符

如果类未定义自己的拷贝赋值运算符，编译器会自动合成一个拷贝赋值运算符

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（通过=delete）

合成拷贝运算符会将右侧运算对象的每个非 static 成员赋予左侧对象的相应成员，这一工作通过成员类型自己的拷贝赋值运算符来完成

一个关于实现拷贝构造函数、拷贝赋值运算符重载的例子



析构函数

调用时机——对象被销毁。只要一个对象被销毁，就会执行其析构函数。

析构函数名字由波浪号接类名组成，没有返回值，也不接受参数。

因为析构不接受参数，所以不能被重载，一个类只能有一个析构函数。不同于构造函数。

析构函数包括一个函数体和一个隐式的析构部分。先执行函数体，然后执行析构部分销毁成员。成员按初始化的顺序逆序销毁。

ps：

1.析构函数函数的函数体部分不做销毁对象的工作（一般做资源释放工作，如释放网络连接、句柄等），在函数体运行之后，在隐式的析构部分销毁对象。析构部分发生的事情完全依赖于成员的类型。

2.内置类型没有析构函数，只有类类型有析构函数。

3.当未定义析构函数时，编译器会定义一个合成析构函数。合成析构函数的函数体是空的。

析构函数调用时机：

1.变量离开作用域时被销毁

2.当一个对象被销毁时，其成员被销毁

3.容器被销毁时，其元素被销毁

4.对于动态分配的对象，对指向它的指针应用delete运算符时被销毁

5，对于临时对象，当创建它的完整表达式结束时被销毁

销毁指针：

内置指针：隐式地销毁一个内置指针类型的成员不会 delete 它所指向的对象。

智能指针：智能指针是类类型，具有析构函数。智能指针的析构函数会递减对象的引用计数，如果计数变为 0，则销毁对象并释放内存。

三/五法则

三个可以控制类的拷贝操作：拷贝构造函数、拷贝赋值运算符、析构函数。

此外拷贝控制成员还有移动构造函数、移动赋值运算符。

ps：C++并不要求一定要定义全部，但这些操作通常应该被看做一个整体，如果定义了其中一个操作，一般也需要定义其他操作。

确定类是否需要自己的拷贝控制成员的两个基本原则：

1.判断它是否需要一个析构函数。如果它需要自定义一个析构函数，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。

2.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。但是需要拷贝构造函数不意味着一定需要析构函数。

使用=default

类似默认构造函数，将拷贝控制成员定义为 =default 可以显式地要求编译器生成合成的版本。

image.png

如果在类内使用 =default，合成的函数将隐式地声明为内联函数。如果不希望合成内联函数，则应该在类外定义处使用 =default。

ps：只能对默认构造函数或拷贝构造成员这些具有合成版本的函数使用 =default。

阻止拷贝

大多数类应该隐式地或显式地定义默认构造函数、拷贝构造函数和拷贝赋值运算符。但是有一些例外需要阻止类进行拷贝或赋值。如 iostream 类、unique\_ptr 等。C++提供了删除一个函数的方法，即在参数列表后添加 =delete

image.png

=delete与=default使用形式上的不同点：

1.=default只能用于可合成的函数，=delete可以用于任何函数

2.=defaule能用在任何函数定义处，=delete只能用在第一次声明函数的时候

ps：析构函数无法删除（不能显式定义为delete），因为析构函数包括一个函数体和一个隐式的析构部分，如果删了就无法释放资源和销毁对象（如果是智能指针，也无法自动释放内存）

在以下情况，类的某些合成的成员会被定义为删除的函数：

1.如果类的某个数据成员的析构函数是删除的或不可访问的（如 private 的），则该类的合成析构函数、合成拷贝构造函数和默认构造函数被定义为删除的（我不理解书上这句话的意思，因为析构函数不能被定义为delete，也无法定义为private）

2.如果类的某个数据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。（如文件句柄、网络连接对象禁止拷贝，单例）

3.如果类的某个数据成员的拷贝赋值运算符是删除的或不可访问的，则类的合成拷贝赋值运算符被定义为删除的。（如文件句柄、网络连接对象禁止拷贝，单例）

4.如果类有一个 const 成员或引用成员，则类的合成拷贝赋值运算符被定义为删除的。理解：因为 const 成员不能修改值，所以不能为它赋值（但是拷贝构造函数在初始化时执行成员初始化列表，是可以的）。

5.如果类有一个没有类内初始化器且未显式定义默认构造函数的 const 成员或没有类内初始化器的引用成员，则该类的默认构造函数被定义为删除的。

ps：总而言之，如果一个类有数据成员不能默认构造、拷贝、复制或销毁，则对应的成员函数被定义为删除的。

旧标准阻止拷贝的方法

将拷贝构造函数和拷贝赋值运算符定义为private，同时为了防止友元调用私有方法进行复制，只将这两个成员函数声明而不定义。

拷贝控制和资源管理

通常管理类外资源的类都需要定义拷贝控制成员，因为它们需要定义析构函数来释放对象所分配的资源。

ps：这里的“类外资源”应该是非托管资源，例如数据库连接、操作系统资源、网络连接、文件句柄等。

通过定义不同的拷贝操作可以实现两种效果：

1.使类的行为看起来像一个值。这种情况拷贝类时副本和原对象是完全独立的。如 strnig 看起来像值（对象完全独立）

2.使类的行为看起来像一个指针。这种情况拷贝类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。如 shared\_ptr 看起来像指针（多个对象共享同一个资源）

还有一些其他的类，如 IO 类型和 unique\_ptr 不允许拷贝和赋值，所以它们的行为既不像值也不像指针。

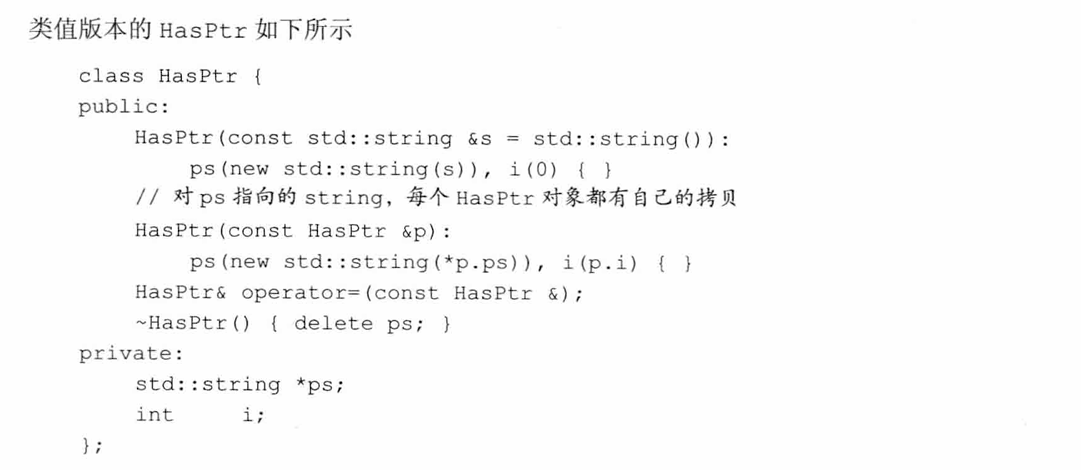
行为像值的类需要什么：

1.拷贝构造函数。定义一个拷贝构造函数，使得每个对象在被复制时创建一个独立的副本，而不是共享相同的资源。

2.拷贝赋值运算符。定义一个拷贝赋值运算符，确保当一个对象赋值给另一个对象时，它们也是独立的。赋值运算符需要释放目标对象当前持有的资源，并复制源对象的资源。

3.析构函数。定义析构函数，以确保对象销毁时释放资源，避免资源泄漏。

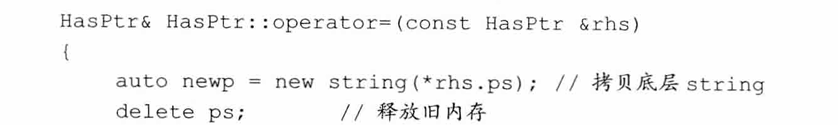
这是书上提供的例子：

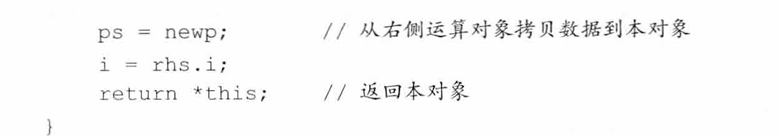


复制运算符通常组合了析构函数和构造函数的操作，在赋值时首先清除左侧运算对象资源，这类似析构函数，随后又从右侧运算对象资源拷贝数据，这类似构造函数。

赋值时，拷贝和资源释放操作按下面这个顺序比较安全：

ps：另一点需要注意的是，如果出现自己赋值给自己的情况，必须保证正常工作（此时释放掉左边资源释放内存后，这个赋值是个未定义行为），我认为加个if(this != &rhs)条件判断（这里必须是地址的比较，因为没有对==和!=运算符进行重载，这个比较是未定义的），在地址不相同时执行下面的释放操作





定义行为像指针的类

对于行为类似指针的类，需要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值。还需要析构函数来释放分配的内存，但是注意不能简单地直接释放关联的内存，应确保最后一个指向该内存的指针也销毁掉后才释放内存。

ps：为了更好地管理内存，令一个类行为像指针的最好方法是使用 shared\_ptr 来管理类内的资源。当然也可以自己实现一个引用计数。

自己实现引用计数定义行为像指针的类：

回顾一下引用计数的要点：

1.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。

2.当创建一个对象时，计数器初始化为 1。

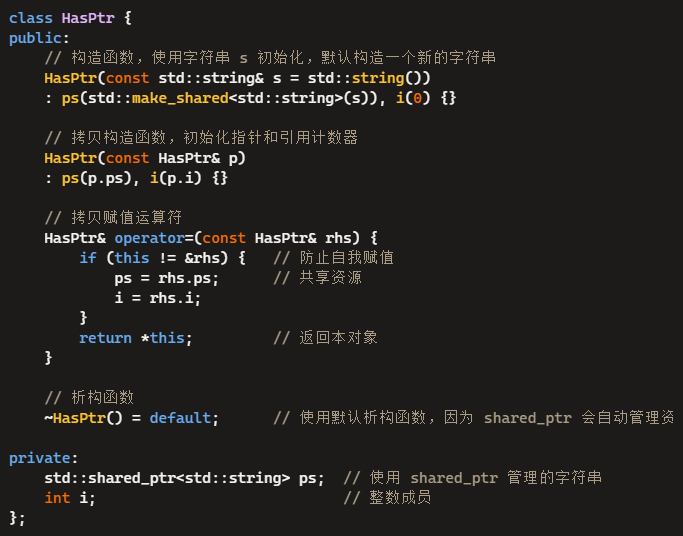
3.拷贝构造函数不创建新的引用计数，而是拷贝对象的计数器并递增它。

4.析构函数递减计数器，如果计数器变为 0，则析构函数释放状态。

5.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 就销毁状态。



使用shared\_ptr的实现

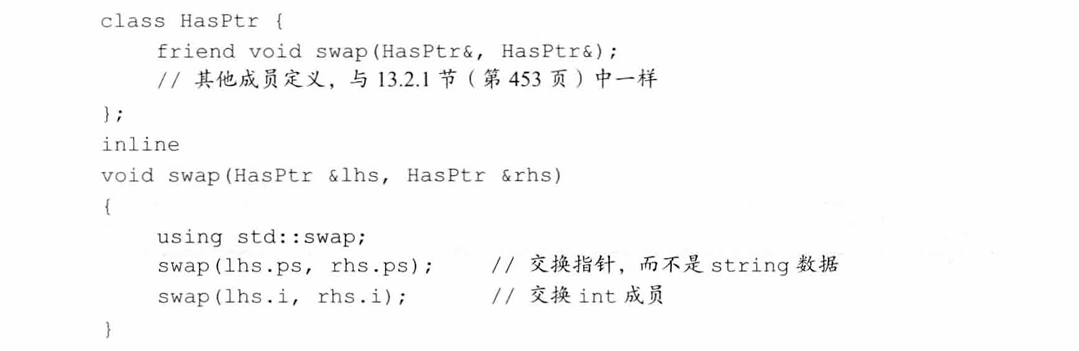


交换操作

除了定义拷贝控制成员，管理资源的类通常还定义一个 swap 函数。因为重排元素顺序的算法在重排元素时，需要使用 swap 来交换元素的位置。因此被重排的元素类型是类时，为类定义自己的 swap 函数非常重要。

ps：swap 不是必要的，但是对于分配了资源的类，定义 swap 有时是一种很重要的优化手段。

下面是书上定义一个swap的例子：



ps：进行一层包装而不直接使用std::swap的原因在于内置类型没有特定版本的swap

swap 函数应该调用 swap，而不是 std::swap

swap 函数用来交换两个类的值，因此实现 swap 函数时要交换类的所有数据成员的值，这也通过 swap 来实现。

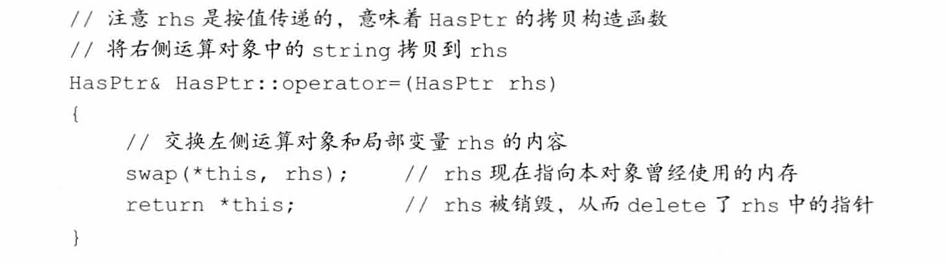
而类的成员可能是另一个类类型，这时为了保证调用的是该类自定义的 swap 版本，应该使用 swap，而不是 std::swap。

ps：上例中的使用方式是正确的，即先 using std::swap，再直接调用 swap。using std::swap 并不会将自定义的 swap 隐藏掉。这保证当某个成员没有自定义的 swap 版本时，能够执行标准库版本

如果一个类定义了自己的 swap，算法将使用自定义版本，否则算法将使用标准库定义的 swap。

在赋值运算符中使用swap

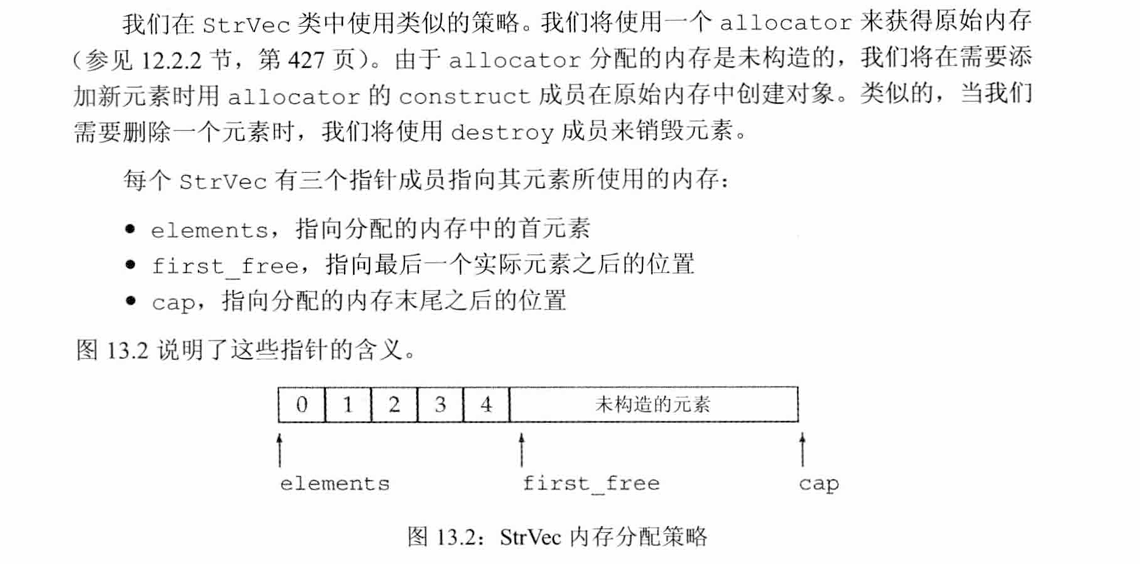
这里的swap接收两个指针

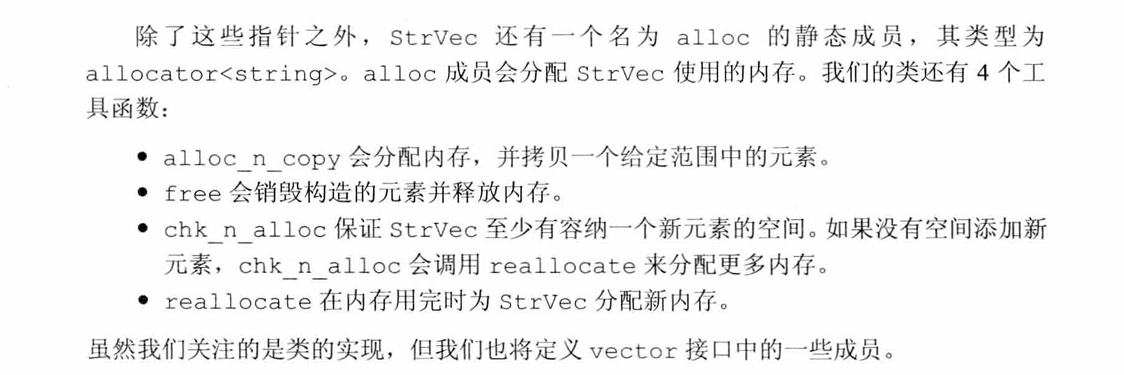


rhs是被拷贝值的一个副本，交换后\*this 持有 rhs 原先指向的地址，而形参rhs为\*this原来指向的空间，在作用域结束后释放这部分内存。这个做法不用if语句块判断是否是自赋值，它原本就是自赋值安全的做法。

动态内存管理类

本书上给了一个动态分配内存的实际例子，使用vector类似的策略完成动态内存的分配，下面是该类的内存模型





对象移动

移动对象的能力是 C++11 中最主要的特性之一。

使用移动的两个常见原因：

1.有时对象拷贝后立即就被销毁了，这是使用移动而非拷贝可以提升性能。

2.如 IO 类或 unique\_ptr 这样的类不能拷贝但可以移动。

 右值引用

右值引用是为了支持移动操作而引入的。

定义：必须绑定到右值的引用，通过 && 操作符来获得右值引用。

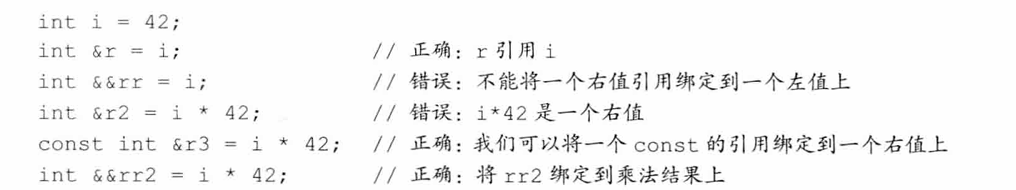
性质：只能绑定到一个将要销毁的对象，且该对象没有其他用户。因此使用右值引用的代码可以自由接管所引用的对象的资源。

左值引用和右值引用

表达式可以分为左值和右值，一般一个左值表达式表示的是一个对象的身份，一个右值表达式表示的对象的值。

左值有持久的状态，而右值要么是字面常量，要么是表达式求值过程中创建的临时对象。因此，左值引用不能绑定到要求转换的表达式、字面常量或返回右值的表达式，而右值引用可以绑定到这些对象，不能直接绑定到一个左值上。

ps：const 的左值引用也可以绑定到右值上。



函返回的左/值的运算符：

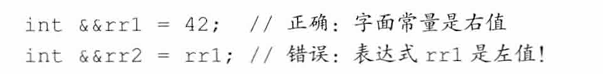
返回左值的运算符：赋值、下标、解引用、前置递增/递减运算符等都返回左值引用（前置递增为左值引用因为递增一个变量而不需要临时变量或额外的步骤）。

返回右值的运算符：算术、关系、位、后置递增/递减运算符等都返回右值。

返回非引用类型的函数返回的也是右值。

变量也可以看作是一个表达式，一个变量表达式是一个左值。

即使一个变量的类型是一个右值引用，但该变量本身也是一个左值。因此不能将一个右值引用直接绑定到一个右值引用变量上。



移动构造函数std::move

定义在头文件utility中

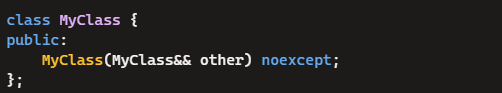
可以通过该方法显式地将一个左值转换为对应的右值引用类型

image.png

但是对一个左值调用move后要保证，不再使用该左值的值，除非销毁它或对它重新赋值。

移动构造函数

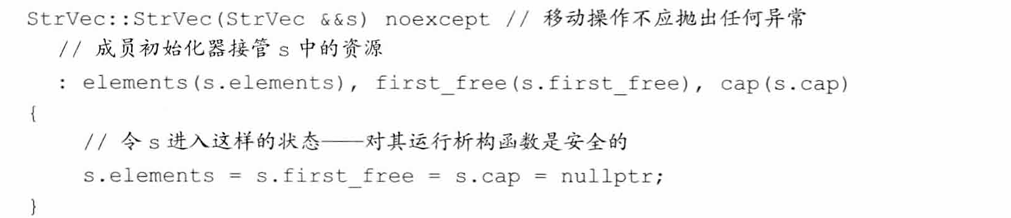
移动构造函数用于从一个即将销毁的对象中“偷取”资源，以构造一个新对象。这样可以避免昂贵的深拷贝操作，提高程序的性能。



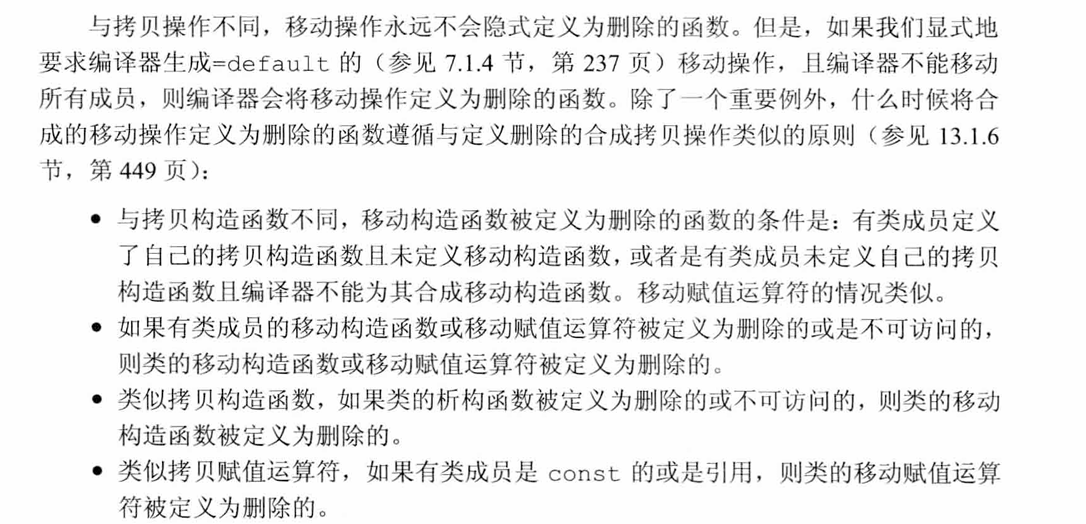
参数只含有一个右值引用

ps：移动构造函数要确保移后源对象是可以直接销毁的。一旦完成资源的移动，源对象必须不再指向被移动的资源。即在执行移动操作之后，源对象（即被移动的对象）的资源（例如内存、文件句柄等）被转移到目标对象中。而源对象本身的指针或者引用必须被置为无效（例如设置为 nullptr），以确保源对象不再持有对原资源的控制或访问权。因此，必须保证移后的源对象可被析构。

下面把源对象中的指针都置为nullpt，不再持有对资源的控制和访问权：

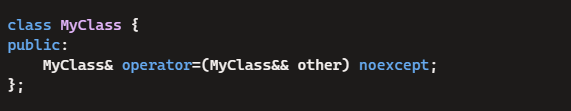


移动操作除了下面的情况，都不会被隐式定义为 =delete 函数



移动赋值运算符

移动赋值运算符用于从一个即将销毁的对象中“偷取”资源，以赋值给当前对象。和移动构造函数一样，移动赋值运算符也能显著提升性能，避免不必要的深拷贝。



ps：声明为noexcept的原因，因为移动操作不会抛出异常，因此直接告诉标准库，可以减少一些额外的检查等操作（我猜的）。

如果一个类定义了拷贝构造函数、拷贝赋值运算符或者析构函数。编译器便不会为它合成移动构造函数和移动赋值函数。

移动迭代器：

包含在头文件#include <iterator> 用于包含 std::move\_iterator 的定义。

创建移动迭代器

移动begin迭代器：std::move\_iterator<std::vector<std::string>::iterator>move\_begin(source.begin()); 创建一个source.begin()移动迭代器。

移动end迭代器：std::move\_iterator<std::vector<std::string>::iterator>move\_end(source.end()); 创建一个source.end() 移动迭代器。