移动构造函数和移动赋值运算符

std::move不负责移动资源，移动资源是移动构造函数和移动赋值运算符定义的。

> 和拷贝构造函数一样，移动构造函数任何额外的参数都需要默认实参。

移动之后，应当保持移后原对象的状态是

- 销毁
- 空

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

以上的rr3，是一个右值引用类型的左值。我们通过调用std::move，就意味着承诺：

- 除了对rr1赋值或销毁它之外，我们将不再使用它
    - 虽然std::move没有移动rr1，rr1上的值依然不变，没有被“移动”，因为其没有定义相应的移动赋值函数。

> 调用move之后，不对移后源的值作任何假设，因为这是移动构造或赋值函数定义的。

定义移动构造函数

```
strvec::strvec(strvec &&s) noexcept // 移动操作不应抛出任何异常
: elements(s.elements),first_free(s.first_free),cap(s.cap){
    // s是移后源，我们应当保证其移后是一个有效的状态
    s.elements=s.first_free=s.cap=nullptr;
}
```

> 移动操作一般会改变源对象的值，所以形参一般不是const

移动操作、标准库容器和异常

由于移动操作不分配新的资源。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。**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。

为什么需要noexcept指明标准库认为我们一个函数不会抛出异常？这是因为，一般情况下，

- 标准库容器能对异常发生时其自身的行为提供保障，如vector可以保证，如果调用push_back时发生异常，vector本身不发生改变。

为什么push_back需要标准库提供保障，这是因为push_back操作可能会导致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新内存中，这是异常发生很常见的情形，如分配内存失败等。

那么重新分配的过程中，使用移动构造函数会怎样？移动一个对象会改变源对象的值，如果重新分配内存的过程中，移动了部分而不是全部元素时，抛出一个异常，就会产生问题。旧空间的移动源元素已经被改变了，而新空间未构造的元素可能尚不存在。此情况下，vector就无法满足自身保持不变的要求。

此情况下，使用拷贝构造函数就可以保证这种要求。重新分配内存的时候，很容易出现异常，因此，为了安全，一般标准库容器都采用拷贝构造函数而不是移动构造函数。

> 如果希望使用移动构造，我们必须显示告知我们的移动构造函数不会抛出异常，可以安全使用。

> ❗ 一定要注意，noexcept不是说该函数一定不会发生异常。该类函数发生抛出异常时，编译器将会直接调用std::terminate方法（而不会捕获该异常，捕获异常就是一种保障机制），中断程序运行，某种程度上抑制了异常的传播和扩散。

更多noexcept可以查询c++11新特性：noexcept特性。

移后源对象必须可析构

移动一个对象，并不会销毁该对象，但有时在移动操作后，源对象可能会被销毁。因此，移动操作应当保证移动后，源对象可以被析构，同时保证源对象是一个有效的状态（当然保证可以被析构也是一种有效的状态），如可以被重新赋值，从而安全的使用。

> 移动操作后，源对象的原来的值不做任何保证，可能是相同的值（即没有改变），也可能置为空值，亦或者是其他的。

合成移动构操作

只有当类内所有数据成员（非静态），都可以被移动时，编译器才会合成移动构造函数或移动赋值运算符。当然，编译器可以移动内置类型的成员（虽然该移动可能什么影响也没有）。

删除的析构操作

如果显示请求编译器生成移动操作函数=default，但是不是所有的成员都可以被移动，那么编译器将会将其定义为删除的。

有以下重要规则：

1. 类成员定义了拷贝构造且未定义移动构造，或者类成员未定义自己的拷贝且编译器无法定义合成移动构造，则移动构造函数将会被定义为删除的。移动赋值运算符是相同的。
2. 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。
3. 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。

```
// assume that Y is a class that defines its own copy constructor but not a move
constructor. Also, the compiler will not synthesize its move construct
struct hasY{
    hasY()=default;
    hasY(hasY &&)=default;  // 显示要求编译器生成一个合成的移动构造函数，但是mem无法移动，因此将其定义为删除的
    Y mem; // hasY will have a deleted move constructor
```

```
};
hasY hy,hy2=std::move(hy);  // error:move constructor is deleted
                            // 如果没有显示请求编译器生成合成的移动操作,则
                            // 该调用 hy2=std::move(hy)
                            // 将会自动调用编译器合成的拷贝构造函数
                            // 删除的函数会影响匹配过程
```

> ! 移动操作永远不会隐式的定义为删除的函数,这是因为移动操作不是必须的,如果一个参数是右值引用,那么它优先匹配移动构造函数,但是如果没有,也会匹配拷贝构造函数。因此,移动操作不是必须的,可以用拷贝构造,拷贝赋值完成相应的工作。故而,如果没有定义移动操作,编译器不会自动合成移动操作(**因此,没必要将其置为删除的**),它会优先选择拷贝构造。

> 只有当所有的拷贝控制成员都没有定义,且所有的数据成员都可以移动构造或移动赋值,编译器才会合成移动构造或移动赋值。

另一个重要规则:
移动操作会影响拷贝操作的合成:如果类定义了一个移动构造函数**或**一个移动赋值运算符,则该类的合成拷贝构造**和**拷贝赋值将会默认定义为删除的。(可能是因为,如果不定义的化,编译器认为该类只允许移动,不需要拷贝?)

移动右值,拷贝左值

- 拷贝接受一个`const Foo &`的参数
    - `const Foo &`可以绑定任何左值,也可以绑定到右值上
- 移动接受`Foo &&`的参数(一般不为`const Foo &&`)
    - 只能绑定到右值上

没有移动构造函数,右值会被匹配拷贝构造函数

用拷贝构造代替移动构造一定是安全的。

一个有趣的例子,展示函数匹配和移动操作间的相互 关系。

拷贝并交换赋值运算符和移动操作

```
class Hasptr{
    public:
        Hasptr(Hasptr &&p)noexcept:ps(p.ps),i(p.i){p.ps=nullptr;}// 移动后,将源对象置为有效的状态
        Hasptr&operator=(Hasptr rhs){
            swap(*this,rhs);
            return *this;
        }   // rhs离开作用域会被销毁
}
```

这其中的`Hasptr&operator=(Hasptr rhs)`，其参数是非引用的，说明实参将会被拷贝初始化。依赖于实参的类型，左值将会使用拷贝构造函数，右值将会使用移动构造函数，即该赋值运算符可以**实现拷贝赋值和移动赋值**两种功能。

```
hp=hp2; // hp2是一个左值;hp2通过拷贝构造函数拷贝到形参rhs
hp=std::move(hp2);  // 移动构造函数移动hp2到形参rhs上
```

> ❗ 一般来说最好自己显示给出五个拷贝控制操作，或者显示定义为删除的，或者给出自定义的版本，不要依赖于编译器合成的。