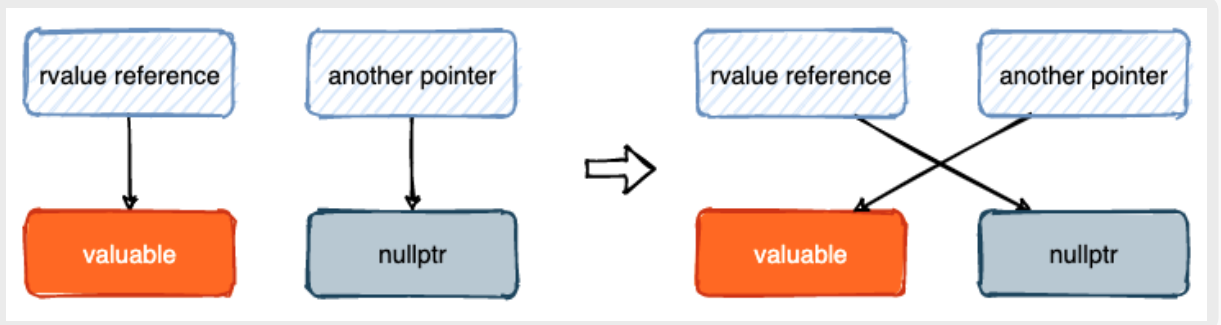


移动构造与移动赋值

移动构造

移动构造函数接收一个右值引用，将需要的内容从该右值引用中“窃取”过来，并让系统自动销毁该“无用”的右值引用

前面已经提到了右值引用所引用的就是一个临时对象，并且随时都有可能被系统回收。那么在该对象被回收之前，我们从中抽取一些有价值的内容放到我们自己这儿，是完全说的过去的



移动构造中的“移动”一词其实并不代表内存的移动，并不是说将右值引用所引用的对象从内存地址 A 搬到另一个引用或指针指向的内存 B，而是移动目标内存的所有权，或者说，交换指针指向

Example

```
class Buz {
private:
    int m_size;
    int *m_array;
public:
    explicit Buz(int size = 4) : m_size(size), m_array(new int[size]) {
        puts("ctor");
    }
    ~Buz() {delete[] m_array;}

    Buz(const Buz& other);           // 拷贝构造

    Buz(Buz&& other) noexcept ;     // 移动构造，接收一个非常量的右值引用
};
```

仍然使用 Buz 这个简单类作为示例

实现

```
Buz::Buz(Buz &&other) noexcept {
    this->m_size = other.m_size;
    this->m_array = other.m_array;

    other.m_array = nullptr;
}
```

过程非常简单，其实就是让 this 中的 m_array 指向右值引用所指向的对象

当右值引用声明周期结束并调用析构函数时，将会 delete 一个空指针，delete[] 空指针 并不会报错

```
Buz buz(10);
for (int i = 0; i < 10; i++) buz[i] = i;
Buz quz = std::move(buz); // 移动构造函数将会被调用
cout << quz[4] << endl;  // 输出结果为 4
```

移动构造与拷贝构造

拷贝构造相当于克隆，调用结束后全局会有两个长的一样的对象；移动构造则类似于传承，父亲传给儿子，那么父亲就没有这东西了，儿子传给孙子，儿子手上就没这东西了，传承物就这一份

如果一个类中定义了移动构造，而没有显式的编写拷贝构造的话，那么拷贝构造将会被禁用

std::unique_ptr 就是这么干的

调用时机

- 1 使用右值直接初始化对象 `Buz buz(10); Buz quz = std::move(buz);`
- 2 当编译器无法使用 RVO 直接在调用者堆栈上构造对象时
 - 首先来看 RVO
 - RVO 即 Return Value Optimization，返回值优化
 - 但一个函数返回对象时，正常情况下是生成返回对象的一个拷贝，再将这个拷贝复制给外部变量，理论上会调用两次拷贝构造函数
 - RVO 就是编译器直接将函数返回的对象构造在调用者的堆栈上，从而避免多次拷贝构造和多次析构函数的调用
 - 在一些复杂情况下，无法使用 RVO 进行优化，那么我们就可以使用移动构造函数，以实现对象的“移动”

关于 noexcept

noexcept 是一个关键字，表示当前函数不会抛出异常，如果函数添加了 noexcept 关键字，并且在运行时抛出了异常，那么程序直接崩溃退出

noexcept 主要是用在 STL 中的容器里面儿，例如 vector。我们知道 vector 是动态的线性容器，当内部存储达到当前存储的容量时，vector 会进行扩容，扩容大小通常是当前的 2 倍

扩容后需要将原有的容器元素迁移到新的更大的内存区域中，这时候如果容器对象所在类中有定义 noexcept 的移动构造函数，那么系统将调用移动构造函数迁移数据，否则将调用拷贝构造

std::move

实现原理

```
template <class _Tp>
inline _LIBCPP_INLINE_VISIBILITY _LIBCPP_CONSTEXPR
typename remove_reference<_Tp>::type&&
move(_Tp&& __t) _NOEXCEPT
{
    typedef _LIBCPP_NODEBUG_TYPE typename remove_reference<_Tp>::type _Up;
    return static_cast<_Up&&>(__t);
}
```

可以看到，形参是一个万能引用，也就是说既能接收左值，也能接收右值。当实参为左值时，_Tp 和 _t 均被推断为左值引用；当实参为右值时，_Tp 将被推断为值类型，_t 为右值引用

remove_reference 的作用就是去除 _Tp 的引用属性，得到 _Tp 的类类型，如 int，Buz 等

```
Buz buz(10);
Buz quz = static_cast<Buz&&>(buz);
```

也会调用移动构造函数，也就是说，std::move 本身只做了类型转换

std::move() 是一个函数模板，其作用就是将一个左值或者是右值强制类型转换成右值，该函数本身并没有任何的移动语义，单纯的 std::move() 调用不会有啥副作用，真正有副作用的是定义在类中的移动构造和移动赋值函数

移动赋值

移动赋值和拷贝赋值一样，也是一个重载函数，重载函数名称为 operator =，也就是赋值运算符，只不过形参为右值引用

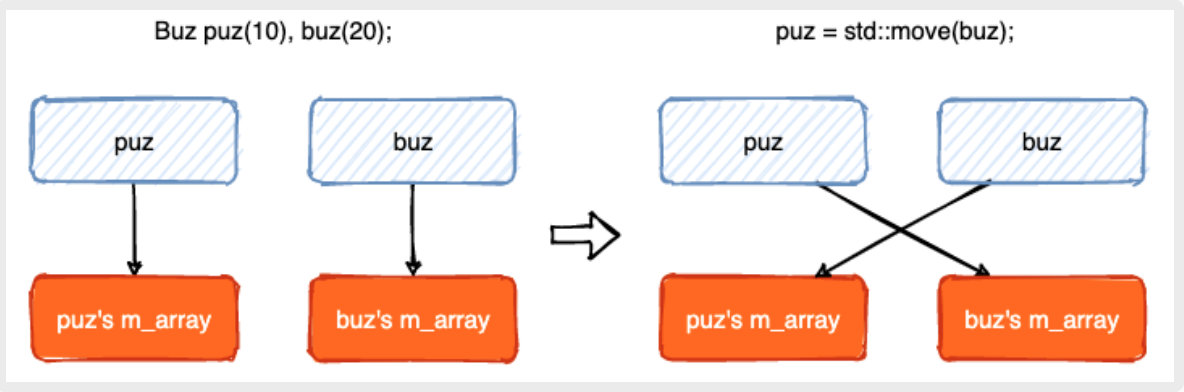
Example

```
Buz &Buz::operator= (Buz &&rhs) {
    using std::swap;

    swap(this->m_size, rhs.m_size);
    swap(this->m_array, rhs.m_array);

    return *this;
}
```

和拷贝赋值函数相比，移动赋值函数中根本不需要临时对象，实参就是临时对象



移动赋值和拷贝赋值发生的时机差不多，都是在赋值语句中调用，只不过此时赋值语句接收一个右值引用而已