4、std::move

Chapter	M
Name	xingzp

一旦开始更规律地使用移动语义，就会开始发现想要调用移动语义，必须处理的对象是l值而不是r值。以下面的交换函数为例:

```
#include <iostream>
#include <string>

template<class T>
void myswap(T& a, T& b)
{
  T tmp { a }; // invokes copy constructor
  a = b; // invokes copy assignment
  b = tmp; // invokes copy assignment
}

int main()
{
  std::string x{ "abc" };
  std::string y{ "de" };

  std::cout << "x: " << x << '\n';
  std::cout << "y: " << y << '\n';

  myswap(x, y);

  std::cout << "x: " << x << '\n';
  std::cout << "y: " << y << '\n';
```

```
  return 0;
}
```

传入两个T类型的对象(在本例中为std::string)时，该函数通过生成三份副本来交换它们的值。因此，这个程序输出:

```
x: abc
y: de
x: de
y: abc
```

正如我们上一课所展示的，复制是低效的。这个版本的swap会生成3个副本。这会导致大量过量的字符串创建和销毁，这是缓慢的。

但是，这里不需要复制。我们真正想做的是交换a和b的值，这可以用3步来完成!因此，如果我们从复制语义切换到移动语义，我们可以使我们的代码更性能。

但如何?这里的问题是形参a和b是l值引用，而不是r值引用，所以我们没有办法调用移动构造函数和移动赋值操作符，而不是调用复制构造函数和复制赋值。默认情况下，我们获得复制构造函数和复制赋值行为。我们该怎么办?

---

std::move

在c++ 11中，std::move是一个标准库函数，它将其参数强制转换(使用static_cast)为一个r-value引用，以便调用move语义。因此，我们可以使用std::move将l值强制转换为更喜欢被移动而不是被复制的类型。std::move在utility头文件中定义。

下面是与上面相同的程序，但是使用了一个myswap()函数，该函数使用std::move将l值转换为r值，因此我们可以调用move语义:

```
#include <iostream>
#include <string>
#include <utility> // for std::move

template<class T>
void myswap(T& a, T& b)
{
  T tmp { std::move(a) }; // invokes move constructor
  a = std::move(b); // invokes move assignment
  b = std::move(tmp); // invokes move assignment
}

int main()
```

```
{
  std::string x{ "abc" };
  std::string y{ "de" };

  std::cout << "x: " << x << '\n';
  std::cout << "y: " << y << '\n';

  myswap(x, y);

  std::cout << "x: " << x << '\n';
  std::cout << "y: " << y << '\n';

  return 0;
}
```

输出的结果与上面相同:

```
x: abc
y: de
x: de
y: abc
```

但它的效率更高。初始化tmp时，我们不复制x，而是使用std::move将l值变量x转换为r值。因为参数是一个r值，所以会调用移动语义，并将x移动到tmp中。

通过更多的交换，变量x的值被移动到y, y的值被移动到x。

另一个例子

在用l值填充容器的元素时，也可以使用std::move，例如std::vector。

在下面的程序中，我们首先使用复制语义向vector添加一个元素。然后使用move语义向vector中添加一个元素。

```
#include <iostream>
#include <string>
#include <utility> // for std::move
#include <vector>

int main()
{
  std::vector<std::string> v;
  std::string str = "Knock";

  std::cout << "Copying str\n";
  v.push_back(str); // calls l-value version of push_back, which copies str into the a
rray element
```

```
  std::cout << "str: " << str << '\n';
  std::cout << "vector: " << v[0] << '\n';

  std::cout << "\nMoving str\n";

  v.push_back(std::move(str)); // calls r-value version of push_back, which moves str
 into the array element

  std::cout << "str: " << str << '\n';
  std::cout << "vector:" << v[0] << ' ' << v[1] << '\n';

  return 0;
}
```

上述程序打印：

```
Copying str
str: Knock
vector: Knock

Moving str
str:
vector: Knock Knock
```

在第一种情况下，我们向push_back()传递了一个l值，因此它使用复制语义向vector添加了一个元素。因此，str中的值保持不变。

在第二种情况下，我们向push_back()传递了一个r值(实际上是通过std::move转换的l值)，因此它使用move语义向vector中添加一个元素。这样效率更高，因为vector元素可以窃取字符串的值，而不必复制它。在本例中，str为空。

此时，值得重申的是，std::move()向编译器提示程序员不再需要该对象了(至少在其当前状态下不再需要)。因此，**您不应该在任何您不想修改的持久对象上使用std::move()，并且您不应该期望任何应用了std::move()的对象的状态在它们被移动后是相同的!**

移动构造和移动赋值函数应该始终将对象保持在定义良好的状态

正如我们在上一课中提到的，总是让被窃取的对象处于某种定义良好的(确定性的)状态是一个好主意。理想情况下，这应该是一个“空状态”。现在我们可以谈谈为什么要这么做:通过std::move，被窃取的对象可能不是一个临时对象。用户可能想再次重用这个(现在是空的)对象，或者以某种方式测试它，用作它途。

在上面的例子中，字符串str在被移动后被设置为空字符串(这是std::string在成功移动后总是做的事情)。如果我们愿意，我们可以重用变量str(或者，如果我们不再需要它，我们可以忽略它)。

在哪些方面std::move是有用的?

std::move在对元素数组进行排序时也很有用。许多排序算法(如选择排序和冒泡排序)通过交换元素对来工作。在前面的课程中，我们不得不求助于复制语义来进行交换。现在我们可以使用移动语义，这更有效。

如果我们想要将一个智能指针管理的内容移动到另一个智能指针上，它也很有用。

结论

为了使l值能调用移动语义而不是复制语义，我们可以使用std::move将l值处理成r值。