**问题背景**

1. #include <iostream>
3. **using** **namespace** std;
5. vector<**int**> doubleValues (**const** vector<**int**>& v)
6. {
7. vector<**int**> new\_values( v.size() );
8. **for** (auto itr = new\_values.begin(), end\_itr = new\_values.end(); itr != end\_itr; ++itr )
9. {
10. new\_values.push\_back( 2 \* \*itr );
11. }
12. **return** new\_values;
13. }
15. **int** main()
16. {
17. vector<**int**> v;
18. **for** ( **int** i = 0; i < 100; i++ )
19. {
20. v.push\_back( i );
21. }
22. v = doubleValues( v );
23. }

先来分析一下上述代码的运行过程。

1. vector<**int**> v;
2. **for** ( **int** i = 0; i < 100; i++ )
3. {
4. v.push\_back( i );
5. }

以上5行语句在栈上新建了一个vector的实例，并在里面放了100个数。

1. v = doubleValues( v )

这条语句调用函数doubleValues,函数的参数类型的const reference，常量引用，那么在实参形参结合的时候并不会将v复制一份，而是直接传递引用。所以在函数体内部使用的v就是刚才创建的那个vector的实例。

但是

1. vector<**int**> new\_values( v.size() );

这条语句新建了一个vector的实例new\_values，并且复制了v的所有内容。但这是合理的，因为我们这是要将一个vector中所有的值翻倍，所以我们不应该改变原有的vector的内容。

1. v = doubleValues( v );

函数执行完之后，new\_values中放了翻倍之后的数值，作为函数的返回值返回。但是注意，这个时候doubleValue(v)的调用已经结束。开始执行 = 的语义。

赋值的过程实际上是将返回的vector<int>复制一份放入新的内存空间，然后改变v的地址，让v指向这篇内存空间。总的来说，我们刚才新建的那个vector又被复制了一遍。

但我们其实希望v能直接得到函数中复制好的那个vector。在C++11之前，我们只能通过传递指针来实现这个目的。但是指针用多了非常不爽。我们希望有更简单的方法。这就是我们为什么要引入右值引用和转移构造函数的原因。

**左值和右值**

在说明左值的定义之前，我们可以先看几个左值的例子。

1. **int** a;
2. a = 1; // here, a is an lvalue

上述的a就是一个左值。

临时变量可以做左值。同样函数的返回值也可以做左值。

1. **int** x;
2. **int**& getRef ()
3. {
4. **return** x;
5. }
7. getRef() = 4;

以上就是函数返回值做左值的例子。

其实左值就是指一个拥有地址的表达式。换句话说，左值指向的是一个稳定的内存空间（即可以是在堆上由用户管理的内存空间，也可以是在栈上，离开了一个block就被销毁的内存空间）。上面第二个例子，getRef返回的就是一个全局变量（建立在堆上），所以可以当做左值使用。

与此相反，右值指向的不是一个稳定的内存空间，而是一个临时的空间。比如说下面的例子：

1. **int** x;
2. **int** getVal ()
3. {
4. **return** x;
5. }
6. getVal();

这里getVal()得到的就是临时的一个值，没法对它进行赋值。  
下面的语句就是错的。

1. getVal() = 1;//compilation error

所以右值只能够用来给其他的左值赋值。

**右值引用**

在C++11中，你可以使用const的左值引用来绑定一个右值，比如说：

1. **const** **int**& val = getVal();//right
2. **int**& val = getVal();//error

因为左值引用并不是左值，并没有建立一片稳定的内存空间，所以如果不是const的话你就可以对它的内容进行修改，而右值又不能进行赋值，所以就会出错。因此只能用const的左值引用来绑定一个右值。

在C++11中，我们可以显示地使用“右值引用”来绑定一个右值,语法是"&&"。因为指定了是右值引用，所以无论是否const都是正确的。

1. **const** string&& name = getName(); // ok
2. string&& name = getName(); // also ok

有了这个功能，我们就可以对原来的左值引用的函数进行重载，重载的函数参数使用右值引用。比如下面这个例子：

1. printReference (**const** String& str)
2. {
3. cout << str;
4. }
6. printReference (String&& str)
7. {
8. cout << str;
9. }

可以这么调用它。

1. string me( "alex" );
2. printReference(  me ); // 调用第一函数，参数为左值常量引用
4. printReference( getName() ); 调用第二个函数，参数为右值引用。

好了，现在我们知道C++11可以进行显示的右值引用了。但是我们如果用它来解决一开始那个复制的问题呢？

这就要引入与此相关的另一个新特性，转移构造函数和转移赋值运算符

**转移构造函数和转移赋值运算符**

假设我们定义了一个ArrayWrapper的类，这个类对数组进行了封装。

1. **class** ArrayWrapper
2. {
3. **public**:
4. ArrayWrapper (**int** n)
5. : \_p\_vals( **new** **int**[ n ] )
6. , \_size( n )
7. {}
8. // copy constructor
9. ArrayWrapper (**const** ArrayWrapper& other)
10. : \_p\_vals( **new** **int**[ other.\_size  ] )
11. , \_size( other.\_size )
12. {
13. **for** ( **int** i = 0; i < \_size; ++i )
14. {
15. \_p\_vals[ i ] = other.\_p\_vals[ i ];
16. }
17. }
18. ~ArrayWrapper ()
19. {
20. **delete** [] \_p\_vals;
21. }
22. **private**:
23. **int** \*\_p\_vals;
24. **int** \_size;
25. };

我们可以看到，这个类的拷贝构造函数显示新建了一片内存空间，然后又对传进来的左值引用进行了复制。

如果传进来的实际参数是一个右值（马上就销毁），我们自然希望能够继续使用这个右值的空间，这样可以节省申请空间和复制的时间。

我们可以使用转移构造函数实现这个功能：

1. **class** ArrayWrapper
2. {
3. **public**:
4. // default constructor produces a moderately sized array
5. ArrayWrapper ()
6. : \_p\_vals( **new** **int**[ 64 ] )
7. , \_size( 64 )
8. {}
10. ArrayWrapper (**int** n)
11. : \_p\_vals( **new** **int**[ n ] )
12. , \_size( n )
13. {}
15. // move constructor
16. ArrayWrapper (ArrayWrapper&& other)
17. : \_p\_vals( other.\_p\_vals  )
18. , \_size( other.\_size )
19. {
20. other.\_p\_vals = NULL;
21. }
23. // copy constructor
24. ArrayWrapper (**const** ArrayWrapper& other)
25. : \_p\_vals( **new** **int**[ other.\_size  ] )
26. , \_size( other.\_size )
27. {
28. **for** ( **int** i = 0; i < \_size; ++i )
29. {
30. \_p\_vals[ i ] = other.\_p\_vals[ i ];
31. }
32. }
33. ~ArrayWrapper ()
34. {
35. **delete** [] \_p\_vals;
36. }
38. **private**:
39. **int** \*\_p\_vals;
40. **int** \_size;
41. };

第一个构造函数就是转移构造函数。它先将other的域复制给自己。尤其是将\_p\_vals的指针赋值给自己的指针，这个过程相当于int的复制，所以非常快。然后将other里面\_p\_vals指针置成NULL。这样做有什么用呢？

我们看到，这个类的析构函数是这样的：

1. ~ArrayWrapper ()
2. {
3. **delete** [] \_p\_vals;
4. }

它会delete掉\_p\_vals的内存空间。但是如果调用析构函数的时候\_p\_vals指向的是NULL，那么就不会delte任何内存空间。

所以假设我们这样使用ArrayWrapper的转移构造函数：

1. ArrayWrapper \*aw = **new** ArrayWrapper((**new** ArrayWrapper(5)));

其中

1. (**new** ArrayWrapper(5)

获得的实例就是一个右值，我们不妨称为r，当整条语句执行结束的时候就会被销毁，执行析构函数。

所以如果转移构造函数中没有

1. other.\_p\_vals = NULL;

的话，虽然aw已经获得了r的\_p\_vals的内存空间，但是之后r就被销毁了，那么r.\_p\_vals的那片内存也被释放了，aw中的\_p\_vals指向的就是一个不合法的内存空间。所以我们就要防止这片空间被销毁。

**右值引用也是左值**

这种说法可能有点绕，来看一个例子：

我们可以定义MetaData类来抽象ArrayWrapper中的数据：

1. **class** MetaData
2. {
3. **public**:
4. MetaData (**int** size, **const** std::string& name)
5. : \_name( name )
6. , \_size( size )
7. {}
9. // copy constructor
10. MetaData (**const** MetaData& other)
11. : \_name( other.\_name )
12. , \_size( other.\_size )
13. {}
15. // move constructor
16. MetaData (MetaData&& other)
17. : \_name( other.\_name )
18. , \_size( other.\_size )
19. {}
21. std::string getName () **const** { **return** \_name; }
22. **int** getSize () **const** { **return** \_size; }
23. **private**:
24. std::string \_name;
25. **int** \_size;
26. };

那么ArrayWrapper类现在就变成这个样子

1. **class** ArrayWrapper
2. {
3. **public**:
4. // default constructor produces a moderately sized array
5. ArrayWrapper ()
6. : \_p\_vals( **new** **int**[ 64 ] )
7. , \_metadata( 64, "ArrayWrapper" )
8. {}
10. ArrayWrapper (**int** n)
11. : \_p\_vals( **new** **int**[ n ] )
12. , \_metadata( n, "ArrayWrapper" )
13. {}
15. // move constructor
16. ArrayWrapper (ArrayWrapper&& other)
17. : \_p\_vals( other.\_p\_vals  )
18. , \_metadata( other.\_metadata )
19. {
20. other.\_p\_vals = NULL;
21. }
23. // copy constructor
24. ArrayWrapper (**const** ArrayWrapper& other)
25. : \_p\_vals( **new** **int**[ other.\_metadata.getSize() ] )
26. , \_metadata( other.\_metadata )
27. {
28. **for** ( **int** i = 0; i < \_metadata.getSize(); ++i )
29. {
30. \_p\_vals[ i ] = other.\_p\_vals[ i ];
31. }
32. }
33. ~ArrayWrapper ()
34. {
35. **delete** [] \_p\_vals;
36. }
37. **private**:
38. **int** \*\_p\_vals;
39. MetaData \_metadata;
40. };

同样，我们使用了转移构造函数来避免代码的复制。但是这里的转移构造函数对吗？

问题出在下面这条语句

1. \_metadata( other.\_metadata )

我们希望的是other.\_metadata是一个右值，然后就会调用MetaData类的转移构造函数来避免数据的复制。但是很可惜，右值引用是左值。

在前面已经说过，左值占用了内存上一片稳定的空间。而右值是一个临时的数据，离开了某条语句就会被销毁。other是一个右值引用，在ArrayWrapper类的转移构造函数的整个作用域中都可以稳定地存在，所以确实占用了内存上的稳定空间，所以是一个左值，因为上述语句调用的并非转移构造函数。所以C++标准库提供了如下函数来解决这个问题：

1. std::move

这条语句可以将左值转换为右值

1. // 转移构造函数
2. ArrayWrapper (ArrayWrapper&& other)
3. : \_p\_vals( other.\_p\_vals  )
4. , \_metadata( std::move( other.\_metadata ) )
5. {
6. other.\_p\_vals = NULL;
7. }

这样就可以避免\_metadata域的复制了。

**函数返回右值引用**

我们可以在函数中显示地返回一个右值引用

1. **int** x;
3. **int** getInt ()
4. {
5. **return** x;
6. }
8. **int** && getRvalueInt ()
9. {
10. // notice that it's fine to move a primitive type--remember, std::move is just a cast
11. **return** std::move( x );
12. }