

03 | 右值和移动究竟解决了什么问题？

2019-12-02 吴咏炜

现代C++实战30讲

[进入课程 >](#)



讲述：吴咏炜

时长 21:44 大小 14.94M



你好，我是吴咏炜。

从上一讲智能指针开始，我们已经或多或少接触了移动语义。本讲我们就完整地讨论一下移动语义和相关的概念。移动语义是 C++11 里引入的一个重要概念；理解这个概念，是理解很多现代 C++ 里的优化的基础。

值分左右

我们常常会说，C++ 里有左值和右值。这话不完全对。标准里的定义实际更复杂，规定了下面这些值类别（value categories）：

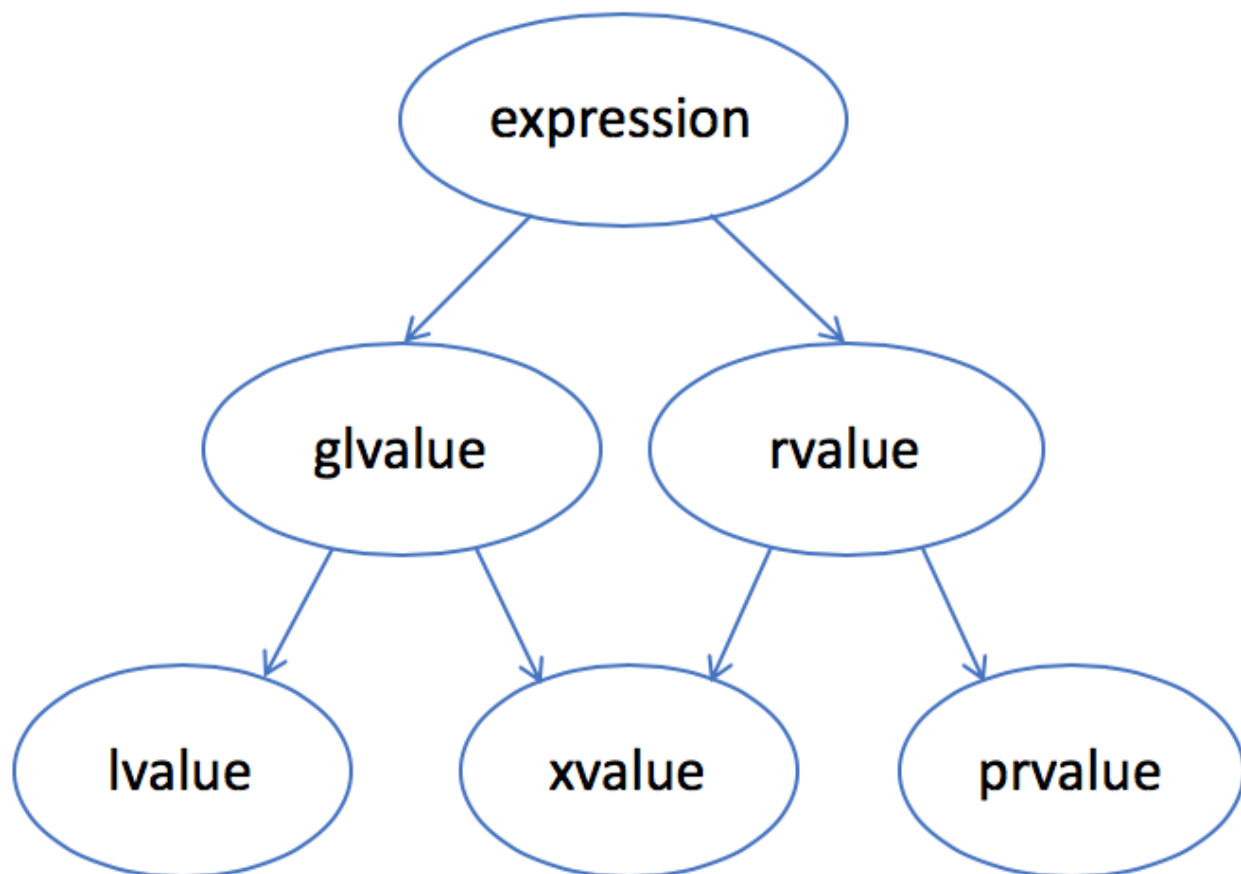


图1: C++ 表达式的值类别

我们先理解一下这些名词的字面含义：

一个 lvalue 是通常可以放在等号左边的表达式，左值

一个 rvalue 是通常只能放在等号右边的表达式，右值

一个 glvalue 是 generalized lvalue，广义左值

一个 xvalue 是 expiring lvalue，将亡值

一个 prvalue 是 pure rvalue，纯右值

还是有点晕，是吧？我们暂且抛开这些概念，只看其中两个：lvalue 和 prvalue。

左值 lvalue 是有标识符、可以取地址的表达式，最常见的情况有：

变量、函数或数据成员的名字

返回左值引用的表达式，如 `++x`、`x = 1`、`cout << ' '`

字符串字面量如 `"hello world"`

在函数调用时，左值可以绑定到左值引用的参数，如 `T&`。一个常量只能绑定到常左值引用，如 `const T&`。

反之，纯右值 `rvalue` 是没有标识符、不可以取地址的表达式，一般也称之为“临时对象”。最常见的情况有：

返回非引用类型的表达式，如 `x++`、`x + 1`、`make_shared<int>(42)`

除字符串字面量之外的字面量，如 `42`、`true`

在 C++11 之前，右值可以绑定到常左值引用（`const lvalue reference`）的参数，如 `const T&`，但不可以绑定到非常左值引用（`non-const lvalue reference`），如 `T&`。从 C++11 开始，C++ 语言里多了一种引用类型——右值引用。右值引用的形式是 `T&&`，比左值引用多一个 `&` 符号。跟左值引用一样，我们可以使用 `const` 和 `volatile` 来进行修饰，但最常见的情况是，我们不会用 `const` 和 `volatile` 来修饰右值。本专栏就属于这种情况。

引入一种额外的引用类型当然增加了语言的复杂性，但也带来了很多优化的可能性。由于 C++ 有重载，我们就可以根据不同的引用类型，来选择不同的重载函数，来完成不同的行为。回想一下，在上一讲中，我们就利用了重载，让 `smart_ptr` 的构造函数可以有不同的行为：

 复制代码

```
1  template <typename U>
2  smart_ptr(const smart_ptr<U>& other) noexcept
3  {
4      ptr_ = other.ptr_;
5      if (ptr_) {
6          other.shared_count_>add_count();
7          shared_count_ =
8              other.shared_count_;
9      }
10 }
11 template <typename U>
12 smart_ptr(smart_ptr<U>&& other) noexcept
```

```
13 {
14     ptr_ = other.ptr_;
15     if (ptr_) {
16         shared_count_ =
17             other.shared_count_;
18         other.ptr_ = nullptr;
19     }
20 }
```

你可能会好奇，使用右值引用的第二个重载函数中的变量 `other` 算是左值还是右值呢？根据定义，`other` 是个变量的名字，变量有标识符、有地址，所以它还是一个左值——虽然它的类型是右值引用。

尤其重要的是，拿这个 `other` 去调用函数时，它匹配的也会是左值引用。也就是说，**类型是右值引用的变量是一个左值**！这点可能有点反直觉，但跟 C++ 的其他方面是一致的。毕竟对于一个右值引用的变量，你是可以取地址的，这点上它和左值完全一致。稍后我们再回到这个话题上来。

再看一下下面的代码：

```
1 smart_ptr<shape> ptr1{new circle()};
2 smart_ptr<shape> ptr2 = std::move(ptr1);
```

 复制代码

第一个表达式里的 `new circle()` 就是一个纯右值；但对于指针，我们通常使用值传递，并不关心它是左值还是右值。

第二个表达式里的 `std::move(ptr)` 就有趣点了。它的作用是把一个左值引用强制转换成一个右值引用，而并不改变其内容。从实用的角度，在我们这儿 `std::move(ptr1)` 等价于 `static_cast<smart_ptr<shape>&&>(ptr1)`。因此，`std::move(ptr1)` 的结果是指向 `ptr1` 的一个右值引用，这样构造 `ptr2` 时就会选择上面第二个重载。

我们可以把 `std::move(ptr1)` 看作是一个有名字的右值。为了跟无名的纯右值 `rvalue` 相区别，C++ 里目前就把这种表达式叫做 `xvalue`。跟左值 `lvalue` 不同，`xvalue` 仍然是不

能取地址的——这点上，xvalue 和 prvalue 相同。所以，xvalue 和 prvalue 都被归为右值 rvalue。我们用下面的图来表示会更清楚一点：

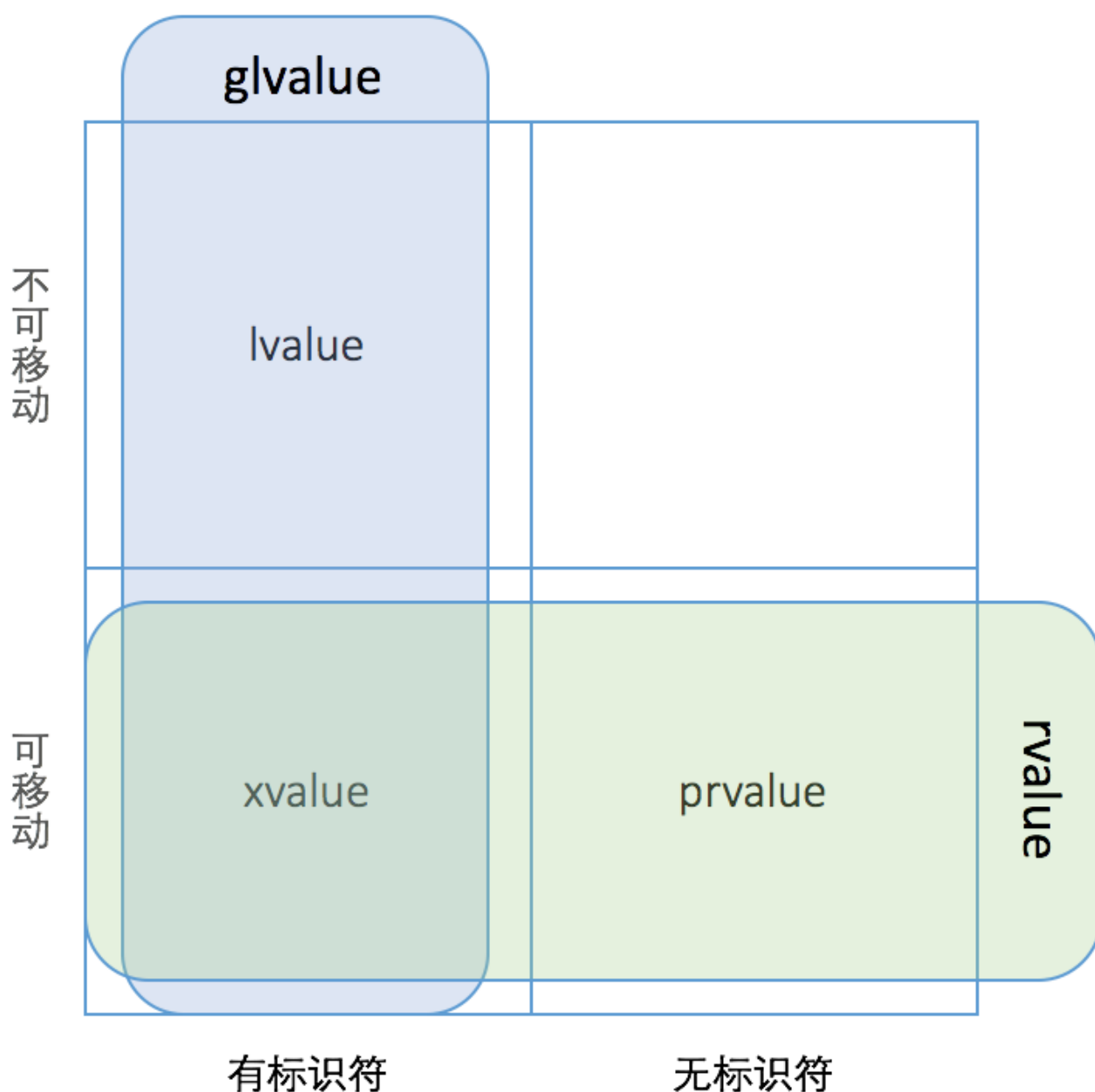



图2：换角度看的表达式值类别

另外请注意，“值类别”（value category）和“值类型”（value type）是两个看似相似、却毫不相干的术语。前者指的是上面这些左值、右值相关的概念，后者则是与引用类型（reference type）相对而言，表明一个变量是代表实际数值，还是引用另外一个数值。在 C++ 里，所有的原生类型、枚举、结构、联合、类都代表值类型，只有引用（&）和指针（*）才是引用类型。在 Java 里，数字等原生类型是值类型，类则属于引用类型。在 Python 里，一切类型都是引用类型。

生命周期和表达式类型

一个变量的生命周期在超出作用域时结束。如果一个变量代表一个对象，当然这个对象的生命周期也在那时结束。那临时对象（prvalue）呢？在这儿，C++ 的规则是：一个临时对象会在包含这个临时对象的完整表达式估值完成后、按生成顺序的逆序被销毁，除非有生命周期延长发生。我们先看一个没有生命周期延长的基本情况：

 复制代码

```
1 process_shape(circle(), triangle());
```

在这儿，我们生成了临时对象，一个圆和一个三角形，它们会在 `process_shape` 执行完成并生成结果对象后被销毁。

我们插入一些实际的代码，就可以演示这一行为：

 复制代码

```
1 #include <stdio.h>
2
3 class shape {
4 public:
5     virtual ~shape() {}
6 };
7
8 class circle : public shape {
9 public:
10     circle() { puts("circle()"); }
11     ~circle() { puts("~circle()"); }
12 };
13
14 class triangle : public shape {
15 public:
16     triangle() { puts("triangle()"); }
17     ~triangle() { puts("~triangle()"); }
18 };
19
20 class result {
21 public:
22     result() { puts("result()"); }
23     ~result() { puts("~result()"); }
24 };
25
26 result
27 process_shape(const shape& shape1,
```

```

28         const shape& shape2)
29     {
30         puts("process_shape()");
31         return result();
32     }
33
34     int main()
35     {
36         puts("main()");
37         process_shape(circle(), triangle());
38         puts("something else");
39     }

```

输出结果可能会是（circle 和 triangle 的顺序在标准中没有规定）：

```

main()
circle()
triangle()
process_shape()
result()
~result()
~triangle()
~circle()
something else

```

目前我让 `process_shape` 也返回了一个结果，这是为了下一步演示的需要。你可以看到结果的临时对象最后生成、最先析构。

为了方便对临时对象的使用，C++ 对临时对象有特殊的生命周期延长规则。这条规则是：

如果一个 prvalue 被绑定到一个引用上，它的使用寿命则会延长到跟这个引用变量一样长。

我们对上面的代码只要改一行就能演示这个效果。把 `process_shape` 那行改成：

```
1 result&& r = process_shape(
```

 复制代码

```
2    circle(), triangle());
```


我们就能看到不同的结果了：

```
main()
circle()
triangle()
process_shape()
result()
~triangle()
~circle()
something else
~result()
```

现在 `result` 的生成还在原来的位置，但析构被延到了 `main` 的最后。

需要万分注意的是，这条生命期延长规则只对 `prvalue` 有效，而对 `xvalue` 无效。如果由于某种原因，`prvalue` 在绑定到引用以前已经变成了 `xvalue`，那生命期就不会延长。不注意这点的话，代码就可能会产生隐秘的 `bug`。比如，我们如果这样改一下代码，结果就不对了：

```
1 #include <utility> // std::move
2 ...
3 result&& r = std::move(process_shape(
4     circle(), triangle()));
```

 复制代码

这时的代码输出就回到了前一种情况。虽然执行到 `something else` 那儿我们仍然有一个有效的变量 `r`，但它指向的对象已经不存在了，对 `r` 的解引用是一个未定义行为。由于 `r` 指向的是栈空间，通常不会立即导致程序崩溃，而会在某些复杂的组合条件下才会引致问题.....

对 C++ 的这条生命期延长规则，在后面讲到视图（view）的时候会十分有用。那时我们会看到，有些 C++ 的用法实际上会隐式地利用这条规则。

此外，参考资料 [5] 中提到了一个有趣的事实：你可以把一个没有虚析构函数的子类对象绑定到基类的引用变量上，这个子类对象的析构仍然是完全正常的——这是因为这条规则只是延后了临时对象的析构而已，不是利用引用计数等复杂的方法，因而只要引用绑定成功，其类型并没有什么影响。

移动的意义

上面我们谈了一些语法知识。就跟学外语的语法一样，这些内容是比较枯燥的。虽然这些知识有时有用，但往往要回过头来看的时候才觉得。初学之时，更重要的是理解为什么，和熟练掌握基本的用法。


对于 `smart_ptr`，我们使用右值引用的目的是实现移动，而实现移动的意义是减少运行的开销——在引用计数指针的场景下，这个开销并不大。移动构造和拷贝构造的差异仅在于：

少了一次 `other.shared_count_>add_count()` 的调用

被移动的指针被清空，因而析构时也少了一次 `shared_count_>reduce_count()` 的调用

在使用容器类的情况下，移动更有意义。我们可以尝试分析一下下面这个假想的语句（假设 `name` 是 `string` 类型）：

```
1 string result =  
2   string("Hello, ") + name + ".";
```


 复制代码

在 C++11 之前的年代里，这种写法是绝对不推荐的。因为它会引入很多额外开销，执行流程大致如下：

1. 调用构造函数 `string(const char*)`，生成临时对象 1；"Hello, " 复制 1 次。

2. 调用 `operator+(const string&, const string&)`, 生成临时对象 2; "Hello, " 复制 2 次, name 复制 1 次。
3. 调用 `operator+(const string&, const char*)`, 生成对象 3; "Hello, " 复制 3 次, name 复制 2 次, "." 复制 1 次。
4. 假设返回值优化能够生效 (最佳情况), 对象 3 可以直接在 `result` 里构造完成。
5. 临时对象 2 析构, 释放指向 `string("Hello, ") + name` 的内存。
6. 临时对象 1 析构, 释放指向 `string("Hello, ")` 的内存。

既然 C++ 是一门追求性能的语言, 一个合格的 C++ 程序员会写:

 复制代码

```
1 string result = "Hello, ";
2 result += name;
3 result += ".";
```

这样的话, 只会调用构造函数一次和 `string::operator+=` 两次, 没有任何临时对象需要生成和析构, 所有的字符串都只复制了一次。但显然代码就啰嗦多了——尤其如果拼接的步骤比较多的话。从 C++11 开始, 这不再是必须的。同样上面那个单行的语句, 执行流程大致如下:


1. 调用构造函数 `string(const char*)`, 生成临时对象 1; "Hello, " 复制 1 次。
2. 调用 `operator+(string&&, const string&)`, 直接在临时对象 1 上面执行追加操作, 并把结果移动到临时对象 2; name 复制 1 次。
3. 调用 `operator+(string&&, const char*)`, 直接在临时对象 2 上面执行追加操作, 并把结果移动到 `result`; "." 复制 1 次。
4. 临时对象 2 析构, 内容已经为空, 不需要释放任何内存。
5. 临时对象 1 析构, 内容已经为空, 不需要释放任何内存。

性能上, 所有的字符串只复制了一次; 虽然比啰嗦的写法仍然要增加临时对象的构造和析构, 但由于这些操作不牵涉到额外的内存分配和释放, 是相当廉价的。程序员只需要牺牲一点点性能, 就可以大大增加代码的可读性。而且, 所谓的性能牺牲, 也只是相对于优化得很

好的 C 或 C++ 代码而言——这样的 C++ 代码的性能仍然完全可以超越 Python 类的语言的相应代码。

此外很关键的一点是，C++ 里的对象缺省都是值语义。在下面这样的代码里：

```
1 class A {  
2     B b_;  
3     C c_;  
4 };
```

 复制代码

从实际内存布局的角度，很多语言——如 Java 和 Python——会在 A 对象里放 B 和 C 的指针（虽然这些语言里本身没有指针的概念）。而 C++ 则会直接把 B 和 C 对象放在 A 的内存空间里。这种行为既是优点也是缺点。说它是优点，是因为它保证了内存访问的局域性，而局域性在现代处理器架构上是绝对具有性能优势的。说它是缺点，是因为复制对象的开销大大增加：在 Java 类语言里复制的是指针，在 C++ 里是完整的对象。这就是为什么 C++ 需要移动语义这一优化，而 Java 类语言里则根本不需要这个概念。

一句话总结，移动语义使得在 C++ 里返回大对象（如容器）的函数和运算符成为现实，因而可以提高代码的简洁性和可读性，提高程序员的生产率。

所有的现代 C++ 的标准容器都针对移动进行了优化。

如何实现移动？

要让你设计的对象支持移动的话，通常需要下面几步：

你的对象应该有分开的拷贝构造和移动构造函数（除非你只打算支持移动，不支持拷贝——如 `unique_ptr`）。

你的对象应该有 `swap` 成员函数，支持和另外一个对象快速交换成员。


在你的对象的名空间下，应当有一个全局的 `swap` 函数，调用成员函数 `swap` 来实现交换。支持这种用法会方便别人（包括你自己在将来）在其他对象里包含你的对象，并快速实现它们的 `swap` 函数。

实现通用的 `operator=`。

上面各个函数如果不抛异常的话，应当标为 `noexcept`。这对移动构造函数尤为重要。

具体写法可以参考我们当前已经实现的 `smart_ptr`：

`smart_ptr` 有拷贝构造和移动构造函数（虽然此处我们的模板构造函数严格来说不算拷贝或移动构造函数）。移动构造函数应当从另一个对象获取资源，清空其资源，并将其置为一个可析构的状态。

 复制代码

```
1 smart_ptr(const smart_ptr& other) noexcept
2 {
3     ptr_ = other.ptr_;
4     if (ptr_) {
5         other.shared_count_
6             ->add_count();
7         shared_count_ =
8             other.shared_count_;
9     }
10 }
11 template <typename U>
12 smart_ptr(const smart_ptr<U>& other) noexcept
13 {
14     ptr_ = other.ptr_;
15     if (ptr_) {
16         other.shared_count_
17             ->add_count();
18         shared_count_ =
19             other.shared_count_;
20     }
21 }
22 template <typename U>
23 smart_ptr(smart_ptr<U>&& other) noexcept
24 {
25     ptr_ = other.ptr_;
26     if (ptr_) {
27         shared_count_ =
28             other.shared_count_;
29         other.ptr_ = nullptr;
30     }
31 }
```

`smart_ptr` 有 `swap` 成员函数。

```
1 void swap(smart_ptr& rhs) noexcept
2 {
3     using std::swap;
4     swap(ptr_, rhs.ptr_);
5     swap(shared_count_,
6         rhs.shared_count_);
7 }
```

[复制代码](#)

有支持 `smart_ptr` 的全局 `swap` 函数。

```
1 template <typename T>
2 void swap(smart_ptr<T>& lhs,
3     smart_ptr<T>& rhs) noexcept
4 {
5     lhs.swap(rhs);
6 }
```

[复制代码](#)

`smart_ptr` 有通用的 `operator=` 成员函数。注意为了避免让人吃惊，通常我们需要将其实现成对 `a = a;` 这样的写法安全。下面的写法算是个小技巧，对传递左值和右值都有效，而且规避了 `if (&rhs != this)` 这样的判断。

```
1 smart_ptr&
2 operator=(smart_ptr rhs) noexcept
3 {
4     rhs.swap(*this);
5     return *this;
6 }
```

[复制代码](#)

不要返回本地变量的引用

有一种常见的 C++ 编程错误，是在函数里返回一个本地对象的引用。由于在函数结束时本地对象即被销毁，返回一个指向本地对象的引用属于未定义行为。理论上来说，程序出任何奇怪的行为都是正常的。

在 C++11 之前，返回一个本地对象意味着这个对象会被拷贝，除非编译器发现可以做返回值优化（named return value optimization，或 NRVO），能把对象直接构造到调用者的

栈上。从 C++11 开始，返回值优化仍可以发生，但在没有返回值优化的情况下，编译器将试图把本地对象移动出去，而不是拷贝出去。这一行为不需要程序员手工用 `std::move` 进行干预——使用 `std::move` 对于移动行为没有帮助，反而会影响返回值优化。

下面是个例子：

 复制代码

```
1 #include <iostream> // std::cout/endl
2 #include <utility>   // std::move
3
4 using namespace std;
5
6 class Obj {
7 public:
8     Obj()
9     {
10         cout << "Obj()" << endl;
11     }
12     Obj(const Obj&)
13     {
14         cout << "Obj(const Obj&)"
15             << endl;
16     }
17     Obj(Obj&&)
18     {
19         cout << "Obj(Obj&&)" << endl;
20     }
21 };
22
23 Obj simple()
24 {
25     Obj obj;
26     // 简单返回对象；一般有 NRVO
27     return obj;
28 }
29
30 Obj simple_with_move()
31 {
32     Obj obj;
33     // move 会禁止 NRVO
34     return std::move(obj);
35 }
36
37 Obj complicated(int n)
38 {
39     Obj obj1;
40     Obj obj2;
41     // 有分支，一般无 NRVO
```



```

42     if (n % 2 == 0) {
43         return obj1;
44     } else {
45         return obj2;
46     }
47 }
48
49 int main()
50 {
51     cout << "*** 1 ***" << endl;
52     auto obj1 = simple();
53     cout << "*** 2 ***" << endl;
54     auto obj2 = simple_with_move();
55     cout << "*** 3 ***" << endl;
56     auto obj3 = complicated(42);
57 }

```

输出通常为：

```

*** 1 ***
Obj ()
*** 2 ***
Obj ()
Obj (Obj &&)
*** 3 ***
Obj ()
Obj ()
Obj (Obj &&)

```

也就是，用了 `std::move` 反而妨碍了返回值优化。

引用坍缩和完美转发

最后讲一个略复杂、但又不得不讲的话题，引用坍缩（又称“引用折叠”）。这个概念在泛型编程中是一定会碰到的。我们今天既然讲了左值和右值引用，也需要一起讲一下。

我们已经讲了对于一个实际的类型 `T`，它的左值引用是 `T&`，右值引用是 `T&&`。那么：

1. 是不是看到 `T&`，就一定是个左值引用？
2. 是不是看到 `T&&`，就一定是个右值引用？

对于前者的回答是“是”，对于后者的回答为“否”。

关键在于，在有模板的代码里，对于类型参数的推导结果可能是引用。我们可以略过一些繁复的语法规则，要点是：

对于 `template <typename T> foo(T&&)` 这样的代码，如果传递过去的参数是左值，`T` 的推导结果是左值引用；如果传递过去的参数是右值，`T` 的推导结果是参数的类型本身。

如果 `T` 是左值引用，那 `T&&` 的结果仍然是左值引用——即 `type& &&` 坍缩成了 `type&`。

如果 `T` 是一个实际类型，那 `T&&` 的结果自然就是一个右值引用。

我们之前提到过，右值引用变量仍然会匹配到左值引用上去。下面的代码会验证这一行为：

 复制代码

```
1 void foo(const shape&)
2 {
3     puts("foo(const shape&)");
4 }
5
6 void foo(shape&&)
7 {
8     puts("foo(shape&&)");
9 }
10
11 void bar(const shape& s)
12 {
13     puts("bar(const shape&)");
14     foo(s);
15 }
16
17 void bar(shape&& s)
18 {
19     puts("bar(shape&&)");
20     foo(s);
21 }
22
```


```
23 int main()
24 {
25     bar(circle());
26 }
```

输出为：

```
bar(shape&&)
foo(const shape&)
```

如果我们要让 `bar` 调用右值引用的那个 `foo` 的重载，我们必须写成：

```
1 foo(std::move(s));
```

 复制代码

或：

```
1 foo(static_cast<shape&&>(s));
```

 复制代码


可如果两个 `bar` 的重载除了调用 `foo` 的方式不一样，其他都差不多的话，我们为什么要提供两个不同的 `bar` 呢？

事实上，很多标准库里的函数，连目标的参数类型都不知道，但我们仍然需要能够保持参数的值类型：左值的仍然是左值，右值的仍然是右值。这个功能在 C++ 标准库中已经提供了，叫 `std::forward`。它和 `std::move` 一样都是利用引用坍缩机制来实现。此处，我们不介绍其实现细节，而是重点展示其用法。我们可以把我们的两个 `bar` 函数简化成：

```
1 template <typename T>
2 void bar(T&& s)
3 {
4     foo(std::forward<T>(s));
5 }
```

 复制代码

对于下面这样的代码：

 复制代码

```
1 circle temp;  
2 bar(temp);  
3 bar(circle());
```

现在的输出是：

```
foo(const shape&)  
foo(shape&&)
```

一切如预期一样。

因为在 `T` 是模板参数时，`T&&` 的作用主要是保持值类别进行转发，它有个名字就叫“转发引用”（forwarding reference）。因为既可以是左值引用，也可以是右值引用，它也曾被叫做“万能引用”（universal reference）。

内容小结

本讲介绍了 C++ 里的值类型，重点介绍了临时变量、右值引用、移动语义和实际的编程用法。由于这是 C++11 里的重点功能，你对于其基本用法需要牢牢掌握。

课后思考

留给你两道思考题：

1. 请查看一下标准函数模板 `make_shared` 的声明，然后想一想，这个函数应该是怎样实现的。
2. 为什么 `smart_ptr::operator=` 对左值和右值都有效，而且不需要对等号两边是否引用同一对象进行判断？

欢迎留言和我交流你的看法，尤其是对第二个问题。

参考资料

[1] cppreference.com, “Value categories” .

🔗 https://en.cppreference.com/w/cpp/language/value_category

[1a] cppreference.com, “值类别” .

🔗 https://zh.cppreference.com/w/cpp/language/value_category

[2] Anders Schau Knatten, “lvalues, rvalues, glvalues, prvalues, xvalues, help!” .

🔗 <https://blog.knatten.org/2018/03/09/lvalues-rvalues-glvalues-prvalues-xvalues-help/>

[3] Jeaye, “Value category cheat-sheet” .

🔗 <https://blog.jeaye.com/2017/03/19/xvalues/>

[4] Thomas Becker, “C++ rvalue references explained” .

🔗 http://thbecker.net/articles/rvalue_references/section_01.html

[5] Herb Sutter, “GotW #88: A candidate for the ‘most important const’ ” .

🔗 <https://herbsutter.com/2008/01/01/gotw-88-a-candidate-for-the-most-important-const/>


点击查看 

打卡学习 C++ 拒绝从入门到放弃



PC 端用户扫码参与



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

上一篇 02 | 自己动手，实现C++的智能指针

精选留言 (2)

写留言



糖

2019-12-02

又是看不懂的一节。。。老师讲的课程太深刻了。。。

1. 本来感觉自己还比较了解左右值的区别，但是，文中提到：一个 lvalue 是通常可以放在等号左边的表达式，左值，然后下面说：字符串字面量如 "hello world"，但字符串字面量貌似不可以放到等号左边，搞晕了。

2. 内存访问的局域性是指什么呢？又有何优势呢？老师能提供介绍的链接吗...

展开 ∨



1



罗乾林

2019-12-02

平时Java是主要使用语言，也来回答一下

1、make_shared 创建(new)新对象根据传入的值类别调用拷贝构造或移动构造,然后将新对象的指针给shared_ptr，其中我看见了_Types&&和forward

2、smart_ptr::operator= 中参数为值传递，会先调用smart_ptr的拷贝构造函数，生成了临时对象，然后调用swap，...

展开 ∨

