C++11移动构造函数详解

在 C++ 11 标准之前（C++ 98/03 标准中），如果想用其它对象初始化一个同类的新对象，只能借助类中的复制（拷贝）构造函数。通过《C++拷贝构造函数》一节的学习我们知道，拷贝构造函数的实现原理很简单，就是为新对象复制一份和其它对象一模一样的数据。

> 需要注意的是，当类中拥有指针类型的成员变量时，拷贝构造函数中需要以深拷贝（而非浅拷贝）的方式复制该指针成员。有关深拷贝和浅拷贝以及它们的区别，读者可阅读《C++深拷贝和浅拷贝》一文做详细了解。

举个例子：

```
01. #include <iostream>
02. using namespace std;
03. 
04. class demo{
05. public:
06.     demo():num(new int(0)){
07.         cout<<"construct!"<<endl;
08.     }
09.     //拷贝构造函数
10.     demo(const demo &d):num(new int(*d.num)){
11.         cout<<"copy construct!"<<endl;
12.     }
13.     ~demo(){
14.         cout<<"class destruct!"<<endl;
15.     }
16. private:
17.     int *num;
18. };
19. 
20. demo get_demo(){
21.     return demo();
22. }
23. 
24. int main(){
25.     demo a = get_demo();
26.     return 0;
27. }
```

> 如上所示，我们为 demo 类自定义了一个拷贝构造函数。该函数在拷贝 d.num 指针成员时，必须采用深拷贝的方式，即拷贝该指针成员本身的同时，还要拷贝指针指向的内存资源。否则一旦多个对象中的指针成员指向同一块堆空间，这些对象析构时就会对该空间释放多次，这是不允许的。

可以看到，程序中定义了一个可返回 demo 对象的 get_demo() 函数，用于在 main() 主函数中初始化 a 对象，其整个初始化的流程包含以下几个阶段：

1. 执行 get_demo() 函数内部的 demo() 语句，即调用 demo 类的默认构造函数生成一个匿名对象；
2. 执行 return demo() 语句，会调用拷贝构造函数复制一份之前生成的匿名对象，并将其作为 get_demo() 函数的返回值（函数体执行完毕之前，匿名对象会被析构销毁）；
3. 执行 a = get_demo() 语句，再调用一次拷贝构造函数，将之前拷贝得到的临时对象复制给 a（此行代码执行完毕，get_demo() 函数返回的对象会被析构）；
4. 程序执行结束前，会自行调用 demo 类的析构函数销毁 a。

注意，目前多数编译器都会对程序中发生的拷贝操作进行优化，因此如果我们使用 VS 2017、codeblocks 等这些编译器运行此程序时，看到的往往是优化后的输出结果：

```
construct!
class destruct!
```

而同样的程序，如果在 Linux 上使用 `g++ demo.cpp -fno-elide-constructors` 命令运行（其中 demo.cpp 是程序文件的名称），就可以看到完整的输出结果：

```
construct!          <-- 执行 demo()
copy construct!     <-- 执行 return demo()
class destruct!     <-- 销毁 demo() 产生的匿名对象
copy construct!     <-- 执行 a = get_demo()
class destruct!     <-- 销毁 get_demo() 返回的临时对象
class destruct!     <-- 销毁 a
```

如上所示，利用拷贝构造函数实现对 a 对象的初始化，底层实际上进行了 2 次拷贝（而且是深拷贝）操作。当然，对于仅申请少量堆空间的临时对象来说，深拷贝的执行效率依旧可以接受，但如果临时对象中的指针成员申请了大量的堆空间，那么 2 次深拷贝操作势必会影响 a 对象初始化的执行效率。

> 事实上，此问题一直存留在以 C++ 98/03 标准编写的 C++ 程序中。由于临时变量的产生、销毁以及发生的拷贝操作本身就是很隐晦的（编译器对这些过程做了专门的优化），且并不会影响程序的正确性，因此很少进入程序员的视野。

那么当类中包含指针类型的成员变量，使用其它对象来初始化同类对象时，怎样才能避免深拷贝导致的效率问题呢？C++11 标准引入了解决方案，该标准中引入了右值引用的语法，借助它可以实现移动语义。

C++移动构造函数（移动语义的具体实现）

所谓移动语义，指的就是以移动而非深拷贝的方式初始化含有指针成员的类对象。简单的理解，移动语义指的就是将其他对象（通常是临时对象）拥有的内存资源“移为已用”。

以前面程序中的 demo 类为例，该类的成员都包含一个整形的指针成员，其默认指向的是容纳一个整形变量的堆空间。当使用 get_demo() 函数返回的临时对象初始化 a 时，我们只需要将临时对象的 num 指针直接浅拷贝给 a.num，然后修改该临时对象中 num 指针的指向（通常另其指向 NULL），这样就完成了 a.num 的初始化。

> 事实上，对于程序执行过程中产生的临时对象，往往只用于传递数据（没有其它的用处），并且会很快会被销毁。因此在使用临时对象初始化新对象时，我们可以将其包含的指针成员指向的内存资源直接移给新对象所有，无需再新拷贝一份，这大大提高了初始化的执行效率。

例如，下面程序对 demo 类进行了修改：

```
#include <iostream>
using namespace std;
class demo{
public:
    demo():num(new int(0)){
        cout<<"construct!"<<endl;
    }

    demo(const demo &d):num(new int(*d.num)){
        cout<<"copy construct!"<<endl;
    }
    //添加移动构造函数
    demo(demo &&d):num(d.num){
        d.num = NULL;
        cout<<"move construct!"<<endl;
    }
    ~demo(){
        cout<<"class destruct!"<<endl;
    }
private:
    int *num;
};
demo get_demo(){
    return demo();
}
int main(){
    demo a = get_demo();
    return 0;
}
```

可以看到，在之前 demo 类的基础上，我们又手动为其添加了一个构造函数。和其它构造函数不同，此构造函数使用右值引用形式的参数，又称为移动构造函数。并且在此构造函数中，num 指针变量采用的是浅拷贝的复制方式，同时在函数内部重置了 d.num，有效避免了“同一块对空间被释放多次”情况的发生。

在 Linux 系统中使用 `g++ demo.cpp -o demo.exe -std=c++0x -fno-elide-constructors` 命令执行此程序，输出结果为：

```
construct!
move construct!
class destruct!
move construct!
class destruct!
class destruct!
```

通过执行结果我们不难得知，当为 demo 类添加移动构造函数之后，使用临时对象初始化 a 对象过程中产生的 2 次拷贝操作，都转由移动构造函数完成。

我们知道，非 const 右值引用只能操作右值，程序执行结果中产生的临时对象（例如函数返回值、lambda 表达式等）既无名称也无法获取其存储地址，所以属于右值。当类中同时包含拷贝构造函数和移动构造函数时，如果使用临时对象初始化当前类的对象，编译器会优先调用移动构造函数来完成此操作。只有当类中没有合适的移动构造函数时，编译器才会退而求其次，调用拷贝构造函数。

> 在实际开发中，通常在类中自定义移动构造函数的同时，会再为其自定义一个适当的拷贝构造函数，由此当用户利用右值初始化类对象时，会调用移动构造函数；使用左值（非右值）初始化类对象时，会调用拷贝构造函数。

读者可能会问，如果使用左值初始化同类对象，但也想调用移动构造函数完成，有没有办法可以实现呢？

默认情况下，左值初始化同类对象只能通过拷贝构造函数完成，如果想调用移动构造函数，则必须使用右值进行初始化。C++11 标准中为了满足用户使用左值初始化同类对象时也通过移动构造函数完成的需求，新引入了 std::move() 函数，它可以将左值强制转换成对应的右值，由此便可以使用移动构造函数。