[C++拷贝构造函数详解](http://blog.csdn.net/lwbeyond/article/details/6202256)

分类： [C/C++](http://blog.csdn.net/lwbeyond/article/category/655963)2011-02-23 13:39 87354人阅读 [评论](http://blog.csdn.net/lwbeyond/article/details/6202256#comments)(143) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/lwbeyond/article/details/6202256#report)

[c++](http://www.csdn.net/tag/c%2b%2b)[fun](http://www.csdn.net/tag/fun)[class](http://www.csdn.net/tag/class)[编译器](http://www.csdn.net/tag/%e7%bc%96%e8%af%91%e5%99%a8)[delete](http://www.csdn.net/tag/delete)[c](http://www.csdn.net/tag/c)

**一. 什么是拷贝构造函数**

首先对于普通类型的对象来说，它们之间的复制是很简单的，例如：

**[c-sharp]** [view plaincopy](http://blog.csdn.net/lwbeyond/article/details/6202256)

1. **int** a = 100;
2. **int** b = a;

而类对象与普通对象不同，类对象内部结构一般较为复杂，存在各种成员变量。  
下面看一个类对象拷贝的简单例子。

**[c-sharp]** [view plaincopy](http://blog.csdn.net/lwbeyond/article/details/6202256)

1. #include <iostream>
2. **using** **namespace** std;
4. **class** CExample {
5. **private**:
6. **int** a;
7. **public**:
8. //构造函数
9. CExample(**int** b)
10. { a = b;}
12. //一般函数
13. **void** Show ()
14. {
15. cout<<a<<endl;
16. }
17. };
19. **int** main()
20. {
21. CExample A(100);
22. CExample B = A; //注意这里的对象初始化要调用拷贝构造函数，而非赋值
23. B.Show ();
24. **return** 0;
25. }

运行程序，屏幕输出100。从以上代码的运行结果可以看出，系统为对象 B 分配了内存并完成了与对象 A 的复制过程。就类对象而言，相同类型的类对象是通过拷贝构造函数来完成整个复制过程的。

下面举例说明拷贝构造函数的工作过程。

**[c-sharp]** [view plaincopy](http://blog.csdn.net/lwbeyond/article/details/6202256)

1. #include <iostream>
2. **using** **namespace** std;
4. **class** CExample {
5. **private**:
6. **int** a;
7. **public**:
8. //构造函数
9. CExample(**int** b)
10. { a = b;}
12. //拷贝构造函数
13. CExample(**const** CExample& C)
14. {
15. a = C.a;
16. }
18. //一般函数
19. **void** Show ()
20. {
21. cout<<a<<endl;
22. }
23. };
25. **int** main()
26. {
27. CExample A(100);
28. CExample B = A; // CExample B(A); 也是一样的
29. B.Show ();
30. **return** 0;
31. }

CExample(const CExample& C)　就是我们自定义的拷贝构造函数。可见，拷贝构造函数是一种**特殊的构造函数**，函数的名称必须和类名称一致，它必须的一个参数是本类型的一个**引用变量**。

**二. 拷贝构造函数的调用时机**

在C++中，下面三种对象需要调用拷贝构造函数！  
**1. 对象以值传递的方式传入函数参数**

**[c-sharp]** [view plaincopy](http://blog.csdn.net/lwbeyond/article/details/6202256)

1. **class CExample**
2. **{**
3. **private:**
4. **int a;**
6. **public:**
7. **//构造函数**
8. **CExample(int b)**
9. **{**
10. **a = b;**
11. **cout<<"creat: "<<a<<endl;**
12. **}**
14. **//拷贝构造**
15. **CExample(const CExample& C)**
16. **{**
17. **a = C.a;**
18. **cout<<"copy"<<endl;**
19. **}**
21. **//析构函数**
22. **~CExample()**
23. **{**
24. **cout<< "delete: "<<a<<endl;**
25. **}**
27. **void Show ()**
28. **{**
29. **cout<<a<<endl;**
30. **}**
31. **};**
33. **//全局函数，传入的是对象**
34. **void g\_Fun(CExample C)**
35. **{**
36. **cout<<"test"<<endl;**
37. **}**
39. **int main()**
40. **{**
41. **CExample test(1);**
42. **//传入对象**
43. **g\_Fun(test);**
45. **return 0;**
46. **}**

调用g\_Fun()时，会产生以下几个重要步骤：  
(1).test对象传入形参时，会先会产生一个临时变量，就叫 C 吧。  
(2).然后调用拷贝构造函数把test的值给C。 整个这两个步骤有点像：CExample C(test);  
(3).等g\_Fun()执行完后, 析构掉 C 对象。

**2. 对象以值传递的方式从函数返回**

**[c-sharp]** [view plaincopy](http://blog.csdn.net/lwbeyond/article/details/6202256)

1. **class CExample**
2. **{**
3. **private:**
4. **int a;**
6. **public:**
7. **//构造函数**
8. **CExample(int b)**
9. **{**
10. **a = b;**
11. **}**
13. **//拷贝构造**
14. **CExample(const CExample& C)**
15. **{**
16. **a = C.a;**
17. **cout<<"copy"<<endl;**
18. **}**
20. **void Show ()**
21. **{**
22. **cout<<a<<endl;**
23. **}**
24. **};**
26. **//全局函数**
27. **CExample g\_Fun()**
28. **{**
29. **CExample temp(0);**
30. **return temp;**
31. **}**
33. **int main()**
34. **{**
35. **g\_Fun();**
36. **return 0;**
37. **}**

当g\_Fun()函数执行到return时，会产生以下几个重要步骤：  
(1). 先会产生一个临时变量，就叫XXXX吧。  
(2). 然后调用拷贝构造函数把temp的值给XXXX。整个这两个步骤有点像：CExample XXXX(temp);  
(3). 在函数执行到最后先析构temp局部变量。  
(4). 等g\_Fun()执行完后再析构掉XXXX对象。

**3. 对象需要通过另外一个对象进行初始化；**

**[c-sharp]** [view plaincopy](http://blog.csdn.net/lwbeyond/article/details/6202256)

1. **CExample A(100);**
2. **CExample B = A;**
3. **// CExample B(A);**

后两句都会调用拷贝构造函数。

**三. 浅拷贝和深拷贝**

**1. 默认拷贝构造函数**

    很多时候在我们都不知道拷贝构造函数的情况下，传递对象给函数参数或者函数返回对象都能很好的进行，这是因为编译器会给我们自动产生一个拷贝构造函数，这就是“默认拷贝构造函数”，这个构造函数很简单，仅仅使用“老对象”的数据成员的值对“新对象”的数据成员一一进行赋值，它一般具有以下形式：

**[c-sharp]** [view plaincopy](http://blog.csdn.net/lwbeyond/article/details/6202256)

1. Rect::Rect(**const** Rect& r)
2. {
3. width = r.width;
4. height = r.height;
5. }

    当然，以上代码不用我们编写，编译器会为我们自动生成。但是如果认为这样就可以解决对象的复制问题，那就错了，让我们来考虑以下一段代码：

**[c-sharp]** [view plaincopy](http://blog.csdn.net/lwbeyond/article/details/6202256)

1. **class** Rect
2. {
3. **public**:
4. Rect()      // 构造函数，计数器加1
5. {
6. count++;
7. }
8. ~Rect()     // 析构函数，计数器减1
9. {
10. count--;
11. }
12. **static** **int** getCount()       // 返回计数器的值
13. {
14. **return** count;
15. }
16. **private**:
17. **int** width;
18. **int** height;
19. **static** **int** count;       // 一静态成员做为计数器
20. };
22. **int** Rect::count = 0;        // 初始化计数器
24. **int** main()
25. {
26. Rect rect1;
27. cout<<"The count of Rect: "<<Rect::getCount()<<endl;
29. Rect rect2(rect1);   // 使用rect1复制rect2，此时应该有两个对象
30. cout<<"The count of Rect: "<<Rect::getCount()<<endl;
32. **return** 0;
33. }

　　这段代码对前面的类，加入了一个静态成员，目的是进行计数。在主函数中，首先创建对象rect1，输出此时的对象个数，然后使用rect1复制出对象rect2，再输出此时的对象个数，按照理解，此时应该有两个对象存在，但实际程序运行时，输出的都是1，反应出只有1个对象。此外，在销毁对象时，由于会调用销毁两个对象，类的析构函数会调用两次，此时的计数器将变为负数。

说白了，就是拷贝构造函数没有处理静态数据成员。

出现这些问题最根本就在于在复制对象时，计数器没有递增，我们重新编写拷贝构造函数，如下：

**[c-sharp]** [view plaincopy](http://blog.csdn.net/lwbeyond/article/details/6202256)

1. **class** Rect
2. {
3. **public**:
4. Rect()      // 构造函数，计数器加1
5. {
6. count++;
7. }
8. Rect(**const** Rect& r)   // 拷贝构造函数
9. {
10. width = r.width;
11. height = r.height;
12. count++;          // 计数器加1
13. }
14. ~Rect()     // 析构函数，计数器减1
15. {
16. count--;
17. }
18. **static** **int** getCount()   // 返回计数器的值
19. {
20. **return** count;
21. }
22. **private**:
23. **int** width;
24. **int** height;
25. **static** **int** count;       // 一静态成员做为计数器
26. };

**2. 浅拷贝**

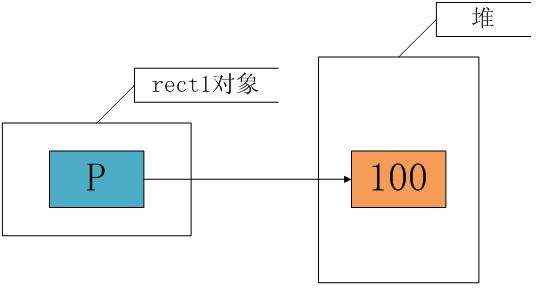
    所谓浅拷贝，指的是在对象复制时，只对对象中的数据成员进行简单的赋值，默认拷贝构造函数执行的也是浅拷贝。大多情况下“浅拷贝”已经能很好地工作了，但是一旦对象存在了动态成员，那么浅拷贝就会出问题了，让我们考虑如下一段代码：

**[c-sharp]** [view plaincopy](http://blog.csdn.net/lwbeyond/article/details/6202256)

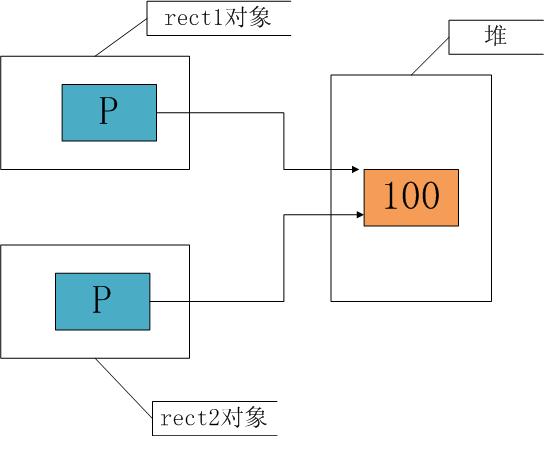
1. **class** Rect
2. {
3. **public**:
4. Rect()      // 构造函数，p指向堆中分配的一空间
5. {
6. p = **new** **int**(100);
7. }
8. ~Rect()     // 析构函数，释放动态分配的空间
9. {
10. **if**(p != NULL)
11. {
12. delete p;
13. }
14. }
15. **private**:
16. **int** width;
17. **int** height;
18. **int** \*p;     // 一指针成员
19. };
21. **int** main()
22. {
23. Rect rect1;
24. Rect rect2(rect1);   // 复制对象
25. **return** 0;
26. }

    在这段代码运行结束之前，会出现一个运行错误。原因就在于在进行对象复制时，对于动态分配的内容没有进行正确的操作。我们来分析一下：

    在运行定义rect1对象后，由于在构造函数中有一个动态分配的语句，因此执行后的内存情况大致如下：



    在使用rect1复制rect2时，由于执行的是浅拷贝，只是将成员的值进行赋值，这时 rect1.p = rect2.p，也即这两个指针指向了堆里的同一个空间，如下图所示：



当然，这不是我们所期望的结果，在销毁对象时，两个对象的析构函数将对同一个内存空间释放两次，这就是错误出现的原因。我们需要的不是两个p有相同的值，而是两个p指向的空间有相同的值，解决办法就是使用“深拷贝”。

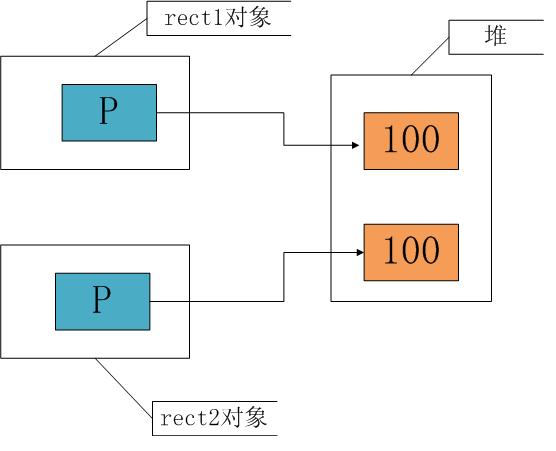
**3. 深拷贝**

    在“深拷贝”的情况下，对于对象中动态成员，就不能仅仅简单地赋值了，而应该重新动态分配空间，如上面的例子就应该按照如下的方式进行处理：

**[c-sharp]** [view plaincopy](http://blog.csdn.net/lwbeyond/article/details/6202256)

1. **class** Rect
2. {
3. **public**:
4. Rect()      // 构造函数，p指向堆中分配的一空间
5. {
6. p = **new** **int**(100);
7. }
8. Rect(**const** Rect& r)
9. {
10. width = r.width;
11. height = r.height;
12. p = **new** **int**;    // 为新对象重新动态分配空间
13. \*p = \*(r.p);
14. }
15. ~Rect()     // 析构函数，释放动态分配的空间
16. {
17. **if**(p != NULL)
18. {
19. delete p;
20. }
21. }
22. **private**:
23. **int** width;
24. **int** height;
25. **int** \*p;     // 一指针成员
26. };

此时，在完成对象的复制后，内存的一个大致情况如下：



此时rect1的p和rect2的p各自指向一段内存空间，但它们指向的空间具有相同的内容，这就是所谓的“深拷贝”。

**3. 防止默认拷贝发生**

    通过对对象复制的分析，我们发现对象的复制大多在进行“值传递”时发生，这里有一个小技巧可以防止按值传递——**声明一个私有拷贝构造函数**。甚至不必去定义这个拷贝构造函数，这样因为拷贝构造函数是私有的，如果用户试图按值传递或函数返回该类对象，将得到一个编译错误，从而可以避免按值传递或返回对象。

**[c-sharp]** [view plaincopy](http://blog.csdn.net/lwbeyond/article/details/6202256)

1. // 防止按值传递
2. **class** CExample
3. {
4. **private**:
5. **int** a;
7. **public**:
8. //构造函数
9. CExample(**int** b)
10. {
11. a = b;
12. cout<<"creat: "<<a<<endl;
13. }
15. **private**:
16. //拷贝构造，只是声明
17. CExample(**const** CExample& C);
19. **public**:
20. ~CExample()
21. {
22. cout<< "delete: "<<a<<endl;
23. }
25. **void** Show ()
26. {
27. cout<<a<<endl;
28. }
29. };
31. //全局函数
32. **void** g\_Fun(CExample C)
33. {
34. cout<<"test"<<endl;
35. }
37. **int** main()
38. {
39. CExample test(1);
40. //g\_Fun(test); 按值传递将出错
42. **return** 0;
43. }

**四. 拷贝构造函数的几个细节**

**1. 拷贝构造函数里能调用private成员变量吗?  
解答：**这个问题是在网上见的，当时一下子有点晕。其时从名子我们就知道拷贝构造函数其时就是一个特殊的**构造函数**，操作的还是自己类的成员变量，所以不受private的限制。

**2. 以下函数哪个是拷贝构造函数,为什么?**

**[c-sharp]** [view plaincopy](http://blog.csdn.net/lwbeyond/article/details/6202256)

1. X::X(**const** X&);
2. X::X(X);
3. X::X(X&, **int** a=1);
4. X::X(X&, **int** a=1, **int** b=2);

**解答：**对于一个类X, 如果一个构造函数的第一个参数是下列之一:  
a) X&  
b) const X&  
c) volatile X&  
d) const volatile X&  
且没有其他参数或其他参数都有默认值,那么这个函数是拷贝构造函数.

**[c-sharp]** [view plaincopy](http://blog.csdn.net/lwbeyond/article/details/6202256)

1. X::X(**const** X&);  //是拷贝构造函数
2. X::X(X&, **int**=1); //是拷贝构造函数
3. X::X(X&, **int** a=1, **int** b=2); //当然也是拷贝构造函数

**3. 一个类中可以存在多于一个的拷贝构造函数吗?  
解答：**类中可以存在超过一个拷贝构造函数。

**[c-sharp]** [view plaincopy](http://blog.csdn.net/lwbeyond/article/details/6202256)

1. **class** X {
2. **public**:
3. X(**const** X&);      // const 的拷贝构造
4. X(X&);            // 非const的拷贝构造
5. };

注意,如果一个类中只存在一个参数为 X& 的拷贝构造函数,那么就不能使用const X或volatile X的对象实行拷贝初始化.

**[c-sharp]** [view plaincopy](http://blog.csdn.net/lwbeyond/article/details/6202256)

1. **class** X {
2. **public**:
3. X();
4. X(X&);
5. };
7. **const** X cx;
8. X x = cx;    // error

如果一个类中没有定义拷贝构造函数,那么编译器会自动产生一个默认的拷贝构造函数。  
这个默认的参数可能为 X::X(const X&)或 X::X(X&),由编译器根据上下文决定选择哪一个。

# [浅谈 C++ 中的 new/delete 和 new[]/delete[]](http://www.cnblogs.com/hazir/p/new_and_delete.html)

在 C++ 中，你也许经常使用 new 和 delete 来动态申请和释放内存，但你可曾想过以下问题呢？

* new 和 delete 是函数吗？
* new [] 和 delete [] 又是什么？什么时候用它们？
* 你知道 operator new 和 operator delete 吗？
* 为什么 new [] 出来的数组有时可以用 delete 释放有时又不行？
* …

如果你对这些问题都有疑问的话，不妨看看我这篇文章。

## new 和 delete 到底是什么？

如果找工作的同学看一些面试的书，我相信都会遇到这样的题：sizeof 不是函数，然后举出一堆的理由来证明 sizeof 不是函数。在这里，和 sizeof 类似，new 和 delete 也不是函数，它们都是 C++ 定义的关键字，通过特定的语法可以组成表达式。和 sizeof 不同的是，sizeof 在编译时候就可以确定其返回值，new 和 delete 背后的机制则比较复杂。  
继续往下之前，请你想想你认为 new 应该要做些什么？也许你第一反应是，new 不就和 C 语言中的 malloc 函数一样嘛，就用来动态申请空间的。你答对了一半，看看下面语句：

string **\***ps **=** **new** string("hello world");

你就可以看出 new 和 malloc 还是有点不同的，malloc 申请完空间之后不会对内存进行必要的初始化，而 new 可以。所以 **new** **expression** 背后要做的事情不是你想象的那么简单。在我用实例来解释 new 背后的机制之前，你需要知道 operator new 和 operator delete 是什么玩意。

## operator new 和 operator delete

这两个其实是 C++ 语言标准库的库函数，原型分别如下：

**void** **\*operator** **new**(**size\_t**); *//allocate an object*

**void** **\*operator** **delete**(**void** **\***); *//free an object*

**void** **\*operator** **new**[](**size\_t**); *//allocate an array*

**void** **\*operator** **delete**[](**void** **\***); *//free an array*

后面两个你可以先不看，后面再介绍。前面两个均是 C++ 标准库函数，你可能会觉得这是函数吗？请不要怀疑，这就是函数！**C++ Primer** 一书上说这不是重载 new 和 delete 表达式（如 operator= 就是重载 **=** 操作符），因为 new 和 delete 是不允许重载的。但我还没搞清楚为什么要用 operator new 和 operator delete 来命名，比较费解。我们只要知道它们的意思就可以了，这两个函数和 C 语言中的 malloc 和 free 函数有点像了，都是用来申请和释放内存的，并且 operator new 申请内存之后不对内存进行初始化，直接返回申请内存的指针。

我们可以直接在我们的程序中使用这几个函数。

## new 和 delete 背后机制

知道上面两个函数之后，我们用一个实例来解释 new 和 delete 背后的机制：

我们不用简单的 C++ 内置类型来举例，使用复杂一点的类类型，定义一个类 A：

**class** **A**

{

public:

A(**int** v) **:** var(v)

{

fopen\_s(**&**file, "test", "r");

}

**~**A()

{

fclose(file);

}

private:

**int** var;

**FILE** **\***file;

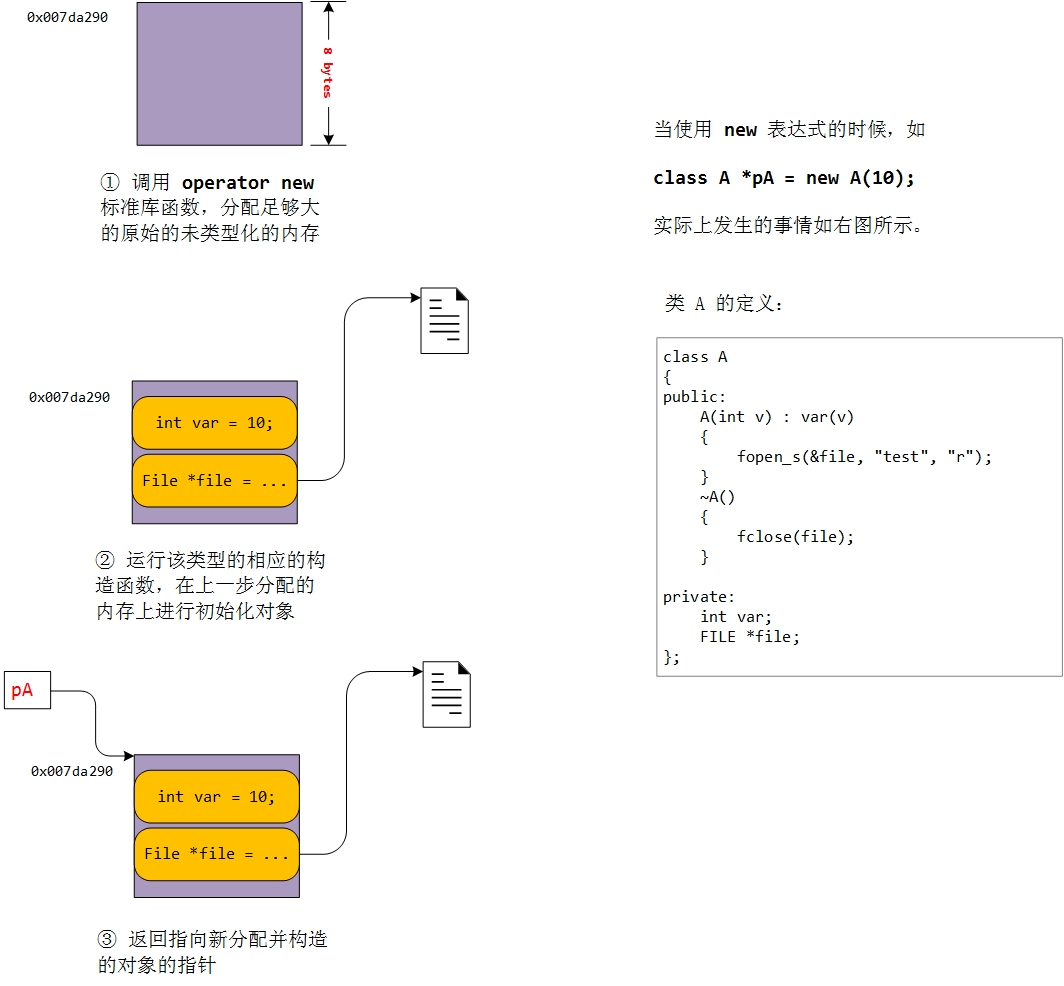
};

很简单，类 A 中有两个私有成员，有一个构造函数和一个析构函数，构造函数中初始化私有变量 var 以及打开一个文件，析构函数关闭打开的文件。

我们使用

**class** A \***pA** **=** **new** A(10);

来创建一个类的对象，返回其指针 pA。如下图所示 new 背后完成的工作：

[](https://github-camo.global.ssl.fastly.net/48182649501319d27121ede912b19be491b1b0a2/687474703a2f2f692e696d6775722e636f6d2f454a736e6b4a342e6a7067)

简单总结一下：

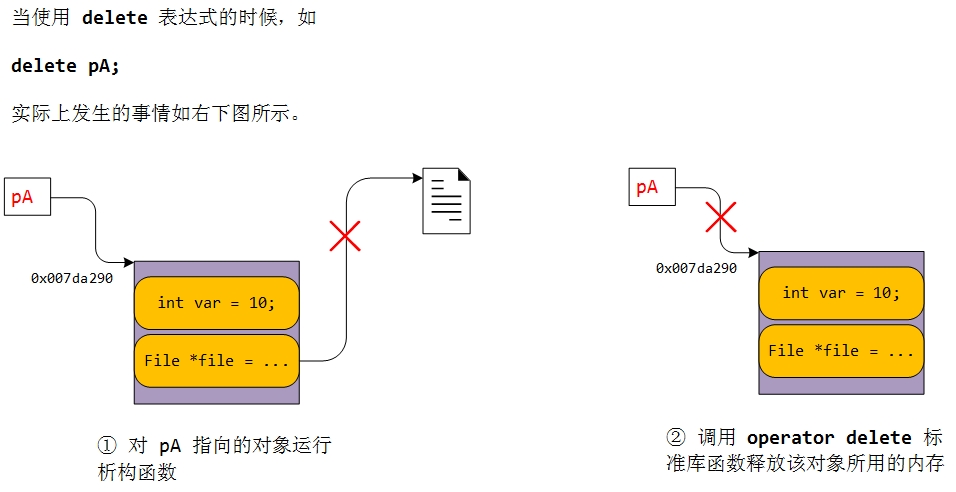
1. 首先需要调用上面提到的 operator new 标准库函数，传入的参数为 class A 的大小，这里为 8 个字节，至于为什么是 8 个字节，你可以看看《深入 C++ 对象模型》一书，这里不做多解释。这样函数返回的是分配内存的起始地址，这里假设是 0x007da290。
2. 上面分配的内存是未初始化的，也是未类型化的，第二步就在这一块原始的内存上对类对象进行初始化，调用的是相应的构造函数，这里是调用 A:A(10); 这个函数，从图中也可以看到对这块申请的内存进行了初始化，var=10, file 指向打开的文件。
3. 最后一步就是返回新分配并构造好的对象的指针，这里 pA 就指向 0x007da290 这块内存，pA 的类型为类 A 对象的指针。

所有这三步，你都可以通过反汇编找到相应的汇编代码，在这里我就不列出了。

好了，那么 delete 都干了什么呢？还是接着上面的例子，如果这时想释放掉申请的类的对象怎么办？当然我们可以使用下面的语句来完成：

**delete** pA;

delete 所做的事情如下图所示：

[](https://github-camo.global.ssl.fastly.net/8bfb4701b5c270dc3b27c86bce57f2d31562d82f/687474703a2f2f692e696d6775722e636f6d2f3156767239467a2e6a7067)

delete 就做了两件事情：

1. 调用 pA 指向对象的析构函数，对打开的文件进行关闭。
2. 通过上面提到的标准库函数 operator delete 来释放该对象的内存，传入函数的参数为 pA 的值，也就是 0x007d290。

好了，解释完了 new 和 delete 背后所做的事情了，是不是觉得也很简单？不就多了一个构造函数和析构函数的调用嘛。

## 如何申请和释放一个数组？

我们经常要用到动态分配一个数组，也许是这样的：

string **\***psa **=** **new** string[10]; *//array of 10 empty strings*

**int** **\***pia **=** **new** **int**[10]; *//array of 10 uninitialized ints*

上面在申请一个数组时都用到了 new [] 这个表达式来完成，按照我们上面讲到的 new 和 delete 知识，第一个数组是 string 类型，分配了保存对象的内存空间之后，将调用 string 类型的默认构造函数依次初始化数组中每个元素；第二个是申请具有内置类型的数组，分配了存储 10 个 int 对象的内存空间，但并没有初始化。

如果我们想释放空间了，可以用下面两条语句：

**delete** [] psa;

**delete** [] pia;

都用到 delete [] 表达式，注意这地方的 [] 一般情况下不能漏掉！我们也可以想象这两个语句分别干了什么：第一个对 10 个 string 对象分别调用析构函数，然后再释放掉为对象分配的所有内存空间；第二个因为是内置类型不存在析构函数，直接释放为 10 个 int 型分配的所有内存空间。

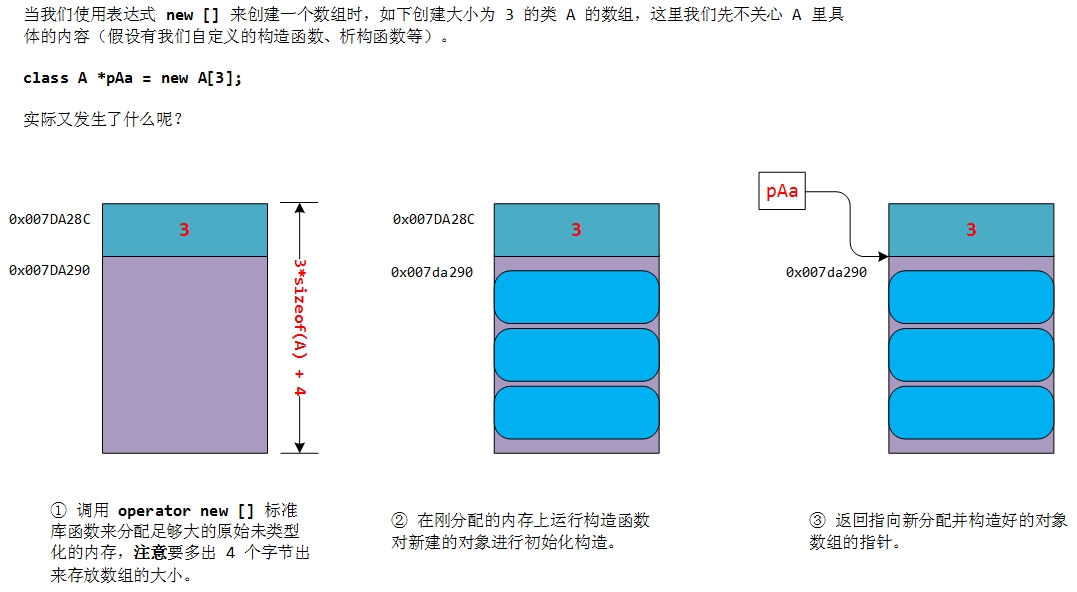
这里对于第一种情况就有一个问题了：**我们如何知道 psa 指向对象的数组的大小？怎么知道调用几次析构函数？**

这个问题直接导致我们需要在 new [] 一个对象数组时，需要保存数组的维度，C++ 的做法是在分配数组空间时多分配了 4 个字节的大小，专门保存数组的大小，在 delete [] 时就可以取出这个保存的数，就知道了需要调用析构函数多少次了。

还是用图来说明比较清楚，我们定义了一个类 A，但不具体描述类的内容，这个类中有显示的构造函数、析构函数等。那么 当我们调用

**class** **A** **\***pAa **=** **new** A[3];

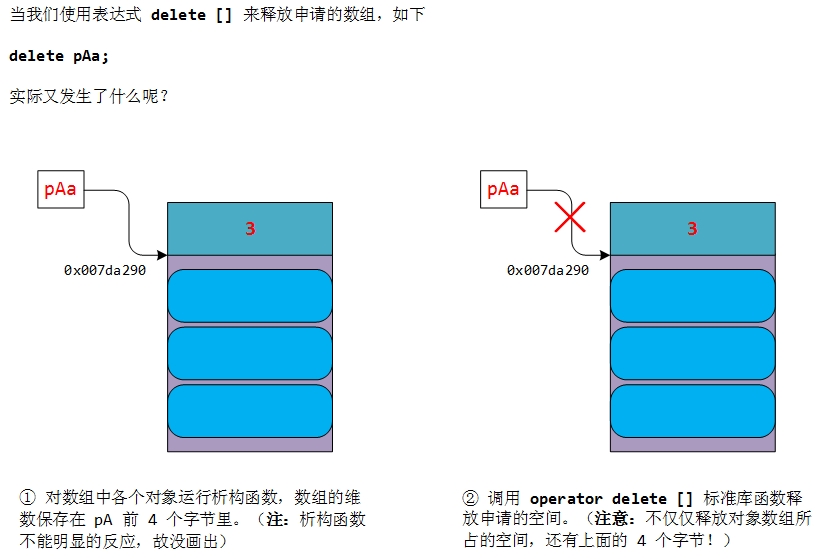
时需要做的事情如下：

[](https://github-camo.global.ssl.fastly.net/1a2da83d54ac0110f8cdb13a6645dd5f9240e760/687474703a2f2f692e696d6775722e636f6d2f366857304431702e6a7067)

从这个图中我们可以看到申请时在数组对象的上面还多分配了 4 个字节用来保存数组的大小，但是最终返回的是对象数组的指针，而不是所有分配空间的起始地址。

这样的话，释放就很简单了：

**delete** []pAa;

[](https://github-camo.global.ssl.fastly.net/db76cce60aaa6f0ad3ab65cd00b0ca26a5271547/687474703a2f2f692e696d6775722e636f6d2f3155425444316c2e6a7067)

这里要注意的两点是：

* 调用析构函数的次数是从数组对象指针前面的 4 个字节中取出；
* 传入 operator delete[] 函数的参数不是数组对象的指针 pAa，而是 pAa 的值减 4。

## 为什么 new/delete 、new []/delete[] 要配对使用？

其实说了这么多，还没到我写这篇文章的最原始意图。从上面解释的你应该懂了 new/delete、new[]/delete[] 的工作原理了，因为它们之间有差别，所以需要配对使用。但偏偏问题不是这么简单，这也是我遇到的问题，如下这段代码：

**int** **\***pia **=** **new** **int**[10];

**delete** []pia;

这肯定是没问题的，但如果把 delete []pia; 换成 delete pia; 的话，会出问题吗？

这就涉及到上面一节没提到的问题了。上面我提到了在 new [] 时多分配 4 个字节的缘由，因为析构时需要知道数组的大小，但如果不调用析构函数呢（如内置类型，这里的 int 数组）？我们在 new [] 时就没必要多分配那 4 个字节， delete [] 时直接到第二步释放为 int 数组分配的空间。如果这里使用 delete pia;那么将会调用 operator delete 函数，传入的参数是分配给数组的起始地址，所做的事情就是释放掉这块内存空间。不存在问题的。

这里说的使用 new [] 用 delete 来释放对象的提前是：对象的类型是内置类型或者是无自定义的析构函数的类类型！

我们看看如果是带有自定义析构函数的类类型，用 new [] 来创建类对象数组，而用 delete 来释放会发生什么？用上面的例子来说明：

**class** **A** **\***pAa **=** **new** **class** **A**[3];

**delete** pAa;

那么 delete pAa; 做了两件事：

* 调用一次 pAa 指向的对象的析构函数；
* 调用 **operator delete(pAa);** 释放内存。

显然，这里只对数组的第一个类对象调用了析构函数，后面的两个对象均没调用析构函数，如果类对象中申请了大量的内存需要在析构函数中释放，而你却在销毁数组对象时少调用了析构函数，这会造成内存泄漏。

上面的问题你如果说没关系的话，那么第二点就是致命的了！直接释放 pAa 指向的内存空间，这个总是会造成严重的段错误，程序必然会奔溃！因为分配的空间的起始地址是 pAa 指向的地方减去 4 个字节的地方。你应该传入参数设为那个地址！

同理，你可以分析如果使用 new 来分配，用 delete [] 来释放会出现什么问题？是不是总会导致程序错误？

总的来说，记住一点即可：**new/delete、new[]/delete[] 要配套使用总是没错的！**

## 参考资料：

C++ Primer 第四版