分享关于自己遇到的一些关于C++内存方面的问题

事情的起因是，在好友系统中，在拉取好友列表时，为了保存一些在向db,switch服务器发包回包后仍然需要使用的信息，用到了player\_t结构体中的session字段。由于好友列表中保存了多个好友信息，自然就想到了用vector数据结构，形式大约如下：

Struct info {

uint32\_t userid;

uint32\_t vip\_level;

……

}；

strct infos {

std::vector<info> friend\_infos;

};

struct infos\* p = (struct infos\*)player->session;

如果struct infos里边都是基本数据类型，这么写是没有问题的（如果结构体不超过session大小），player是new出的对象，内存分配在堆上，session是一个4096个元素的char 数组，内存也分配在堆上，这个操作会把堆上的内存转化会指定的类型；但是当这里不是基本数据类型时，问题就有些模糊不清了。

C++内存分为几个区，堆，栈，全局和常量。

当做了上边操作后， session中的前24个字节被转换为了vector的类型（不同环境下大小可能不同），但是当vector做了push\_back操作后， 被push\_back入vector的元素会存在了哪里呢？ 显然元素用的并不是session的内存。

那用的是哪里的内存呢。

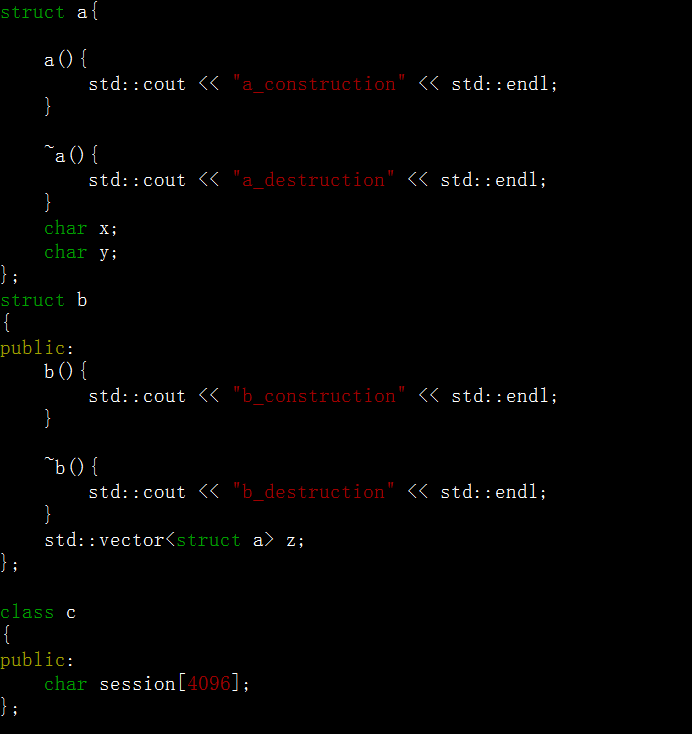
肯定不是全局区和常量区的内存，也不会是栈上的内存，因为并没有一个作用域，超出之后vector里的元素就会消失。

那就只能是vector的实现的时候，在堆的其他地方申请了一块内存。既然内存在堆上，就需要主动的去释放内存。vector在析构时，会销毁其中的元素。（This destroys all container elements, and deallocates all the storage capacity allocated by the vector using its allocator）[**http://www.cplusplus.com/reference/vector/vector/~vector/**](http://www.cplusplus.com/reference/vector/vector/~vector/)

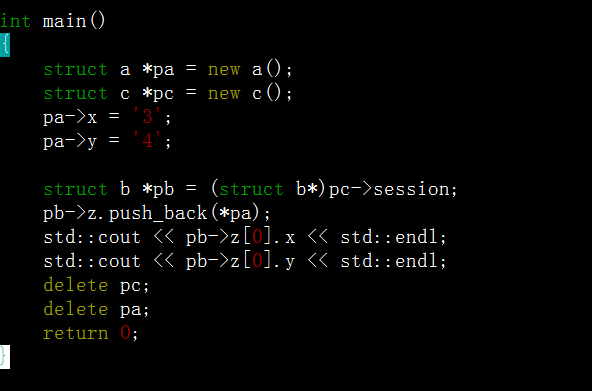
然而由于本例中的vector的结构是从char数组强转来的，析构函数是否会被调用到呢？

本人写了一个小程序做了一个简单的实验：

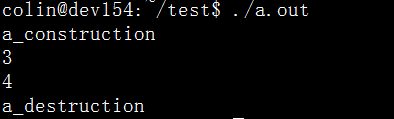
**数据定义：**



**代码：**



**运行结果：**

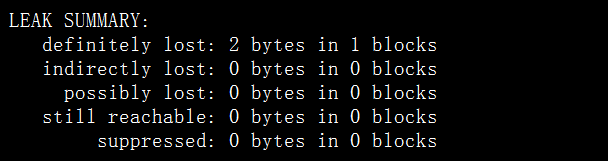


从上述可以看出，虽然显式的用delete pc删除掉了c类型对象，但是并没有触发b类型调用析构函数，也没有触发vector中的a类型调用析构函数（结果中的a\_desturction是main中手动new的pa指向的对象析构时打印的信息）。 就这样，虽然vector的结构不存在了，但是他里边存放元素的内存还没有释放，从来就内存泄露了。

为了确认这是一个内存泄露，可以用valgrind做一个检测：

valgrind --tool=memcheck --leak-check=full ./a.out

结果如下：



很显然，泄露的就是一个a结构体的大小（2字节），就是push\_back进入vector的一个元素没有释放。

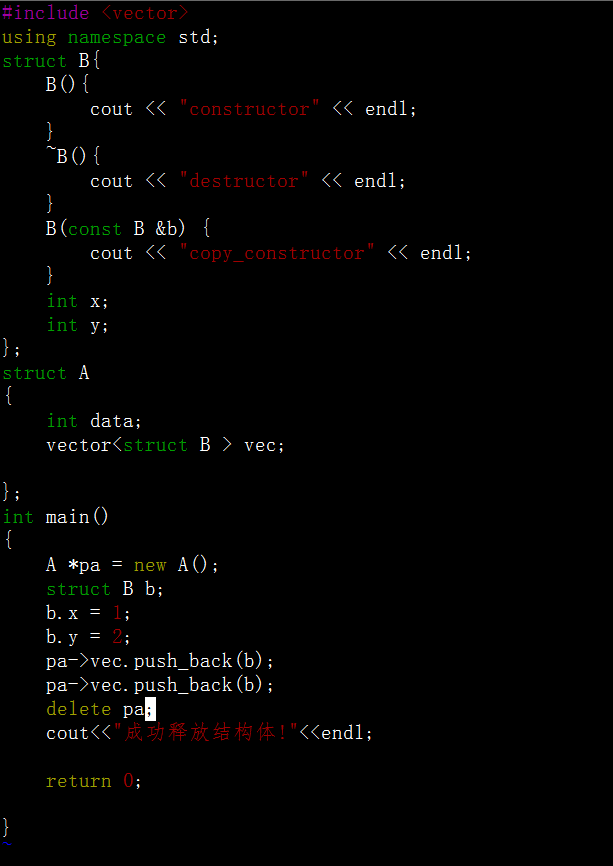
我对此的理解是，把一块内存强转为一个非基本类型，这块内存具有了该类型的结构（有了它的属性和方法），但是并没有创建一个对象（上边代码在强转session时并没有调用构造函数）， 从而不会调用构造函数和析构函数。

在了解这一点后，将代码改为了将session改为强转成一个结构体数组，当然结构体中的成员变量都是基本类型。

总结：在写一种自己没有写过的编码方式时，要多去思考这种写法是否可取，是否有问题，没什么没有见过别人这么写。C++没有垃圾回收器，所以在处理C++内存时要格外小气，尤其在堆上分配内存后，一定要记得回收。

另外

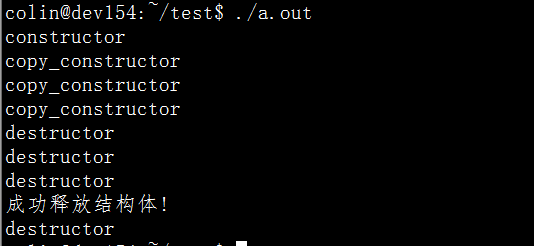
附一个在解决这个问题过程中，碰到的一个其他问题：



这个程序会调用多少次构造函数，拷贝构造函数和析构函数呢。

看起来应该是构造函数一次，拷贝构造函数两次，析构函数三次：

然而运行结果却是：



拷贝构造函数和析构函数各多了一次。

经过和他人讨论和上网搜索得知，这和vector的实现有关。

Vector是动态数组，给元素分配内存时，用的是 1 - 2 -4 -8 -16 ….的方式分配的。

就是说，当push第一个元素时，vector只分配了一个元素大小的内存，存入了第一个对象b， 这时调用了一次拷贝构造函数； 第二次push时，发现内存不够用了，就重新分配了一个两个元素大小的内存，不但将第二个b对象push入新分配内存，还将第一个对象b再从旧内存拷贝的新的内存上边，然后销毁旧内存上边的对象b, 所以，第二次push调用了两次构造函数和一次析构函数。这就是上边多了一次构造函数和一次析构函数的由来。