第23篇-戏说C++ 内存泄漏(A)

铁甲万能狗

自由开发者，专攻C++/Python后端开发(简书平台同号)

阅读该文章，我希望你对重载new操作符和delete操作符有所了解，这是一篇预备文章：《重载C ++中的new和delete运算符》，因为涉及C++的堆内存管理需要这些预备知识。

我们前面两篇随笔已经详细讲述了如何正确地使用 new、new[]、delete、delete[] 操作符去管理堆内存,有了前面的基础，我们本篇专门讨论一些内存泄漏这个问题，我们前面说过“**使用new操作符分配内存而事后忘记使用delete()函数和或delete[]操作符释放该内存块**”这个原因只是一般性，而我们该原因发生的各种情况。

示例导入

我们用《重载C ++中的new和delete运算符》中类似的示例，作为本文展开的话题，首先我们并没有按照我以前所说的那样将类接口定义和类实现分开两个文件来组织代码，是**为了简化我们组织代码的篇幅**，因此该代码的瑕疵之一.我们下面会逐步改正该代码上存在有关内存泄漏上的安全问题-幽灵指针。

```
#include <string>
#include <iostream>
#include<string.h>
class Person{
    double d_height;
    size_t d_age;
    std::string d_name;

public:
        Person(std::string& name,
                    size_t age,double height)
        :d_name(name),d_age(age),d_height(height)
        {}

        void show(){
                std::cout<<"姓名:"<<d_name<<std::endl;
                std::cout<<"年龄:"<<d_age<<std::endl;
                std::cout<<"身高:"<<d_height<<std::endl;
                std::cout<<"身份证:"<<d_idNo<<std::endl;
        }

        void set_height(double height){
            if(height>=0){d_height=height;}
        }
};
```

然后，我们这里有个可以修改Person对象的业务函数，如下面例子的do_something函数，它在处理完Person对象后会释放该Person类型的参数所持有的堆内存,但这个函数设计是存在很多不足的，如果你能够罗列出多于下面5点的话，你对内存泄漏的认识也有一个基本的了解了。

```
    (*s)->show();
    delete (*s);
}
```

幽灵指针

幽灵指针,又叫**悬空指针**、**迷途指针**是指已经被delete或delete[]释放了其所指向堆内存的指针,但它仍然持有**原本堆内存的地址**，并且可能会被程序猿无意间再次使用该变量，可能会产生不可预知的后果。

备注:如果你已经阅读了《深刻理解C指针》这本书的话，你就会发现，我这里阐述的其实就是它“迷途指针”的另外一个C++版本.

产生幽灵指针的场景

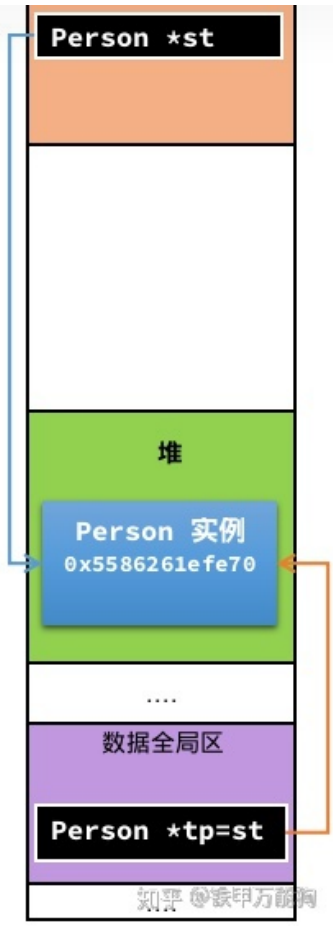
在程序初始化的时候,用new初始化了一个类对象的指针变量,同时又将该指针变量的赋值给另外一个同类类型的全局指针的副本。当然这是一个非常大的安全隐含:“**一个全局的指针对其他类和函数同样可见,并被任意修改**”。因为幽灵指针的危害性是比较隐蔽的，我们这里就需要如此糟糕的例子展示这种指针的危害性。

```
...
int var_init(){
    ....
    Person *st=new Person("艾可",19,87.5);
    //可能本来意图是将该指针缓存到全局变量中，
    //然后等到在其他函数中再次调用
    global Person *tp=st; //tmp是全局指针副本

    //注意：这里修改了一次Person对象
    do_something(&st,73);
    ...
    return 0;
}
```

如上代码在**未执行到do_something之前**,产生有两个Person类型的指针变量指向堆内存中的同一个Person对象
的指针，如下图所示

- 一个位于局部变量st指针:我们知道
- 一个位于全局变量区的pt指针



假设，在某个do_other函数位置被执行了一次,这次使用全局Person对象指针pt，再次修改，其实到了这里，你可以思考一下这段代码存在那些设计上的缺陷!

```
int do_other(){
    std::cout<<"st指向的堆内存地址:"<<st<<std::endl;
    do_something(&pt,72);

    //这里中间假设有很多业务代码.....
    .....
    //这里,程序员不为意再次调用刚才已经执行内存释放的指针
}
```

这里导出了5个错误的设计问题

- 问题1:滥用了全局变量,至少使用一个指向堆内存的指针是一个非常错误的选择。
- 问题2:do_something函数莫名其妙地调用delete操作符号。
- 问题3:已经被delete操作的指针变量,它仍然持有指向**堆**中的**已被回收的内存块的内存地址**。
- 问题4:同一个指针变量两次被delete操作,这会给C++堆内存管理器带来困扰。
- 问题5:**已被回收的堆内存区中存储的用户数据,并没有我们想像中已被销毁**,对于讲究数据安全的应用来说，假设我们在该堆内存区中保存了用名ID或密码这是一个严重的安全隐患。

那么我们如何解决上面提到的问题呢?

- **对于问题1**,有时我们确实需要跨函数调用指向同一个堆内存块的对象数据，并且也不知道会有多少个函数调用会用到**该堆内存块**上的对象数据的时候，明智的选择是使用哈希表(Hash Table)的数据结构，装载初始化那些指向栈内存的指针副本，对于C++来说，可能是这样的哈希声明 std::map<Person*,bool> perMap,**键**部份用来装载Person对象指针,**值**部分用来装载一个布尔标记,用

Person对象指针被delete操作符回收内存了就得设定值部分为**true**。

- **对于问题2**，这个我们已经在内存《第4篇-C++ 的内存回收》我谈论的很清楚
- 首先针对**问题3**题和**问题4**，我们应该**在delete操作后的指针变量重置为nullptr，并且在修改类类型的指针变量指向的对象数据之前，应该做必要的nullptr检测。**
- 问题5,C++编译器设计默认的delete或delete[]操作符的目地仅是告知其内部的堆管理器对指针变量所指向的那**堆内某个内存块**重新打上"已回收"类似的标记,堆内部可能存在很多这样的零散的闲置内存块，操作系统会根据自身的内存管理策略将零散的堆内存块并装成连续的内存块,对以便在其他应用程序需要从堆内申请内存,堆管理恰好就可以将已回收的内存块重新分配给别的应用程序使用，但delete操作并没有被设计为将释放后的内存块清零操作(也就是将该内存块内的所有位重置为0)，是因为这样会加重占用CPU的负担，这并不是C++堆管理器的设计初衷。我们下文会根据问题3,4,5给出一个可行的改善方案。

幽灵指针的风险

注意:就是Person对象指针被delete操作后，只要该指针还未被重置为nullptr,仍然可以访问已被回收的堆内存块，

- 如果已被回收的堆空间存在敏感数据,那么你的程序存在数据泄漏的风险。
- 如果恰逢已被回收的堆空间，操作系统已经分配给其他进程或同一进程的其他线程中的函数使用，那么再次使用该对象指针访问该内存，可能造成程序的数据完整性问题，或者导致程序崩溃。

其实设计良好的类内存回收的代码，可能都需要重载delete操作符或delete[]操作符，本例中Person类内部重载delete操作符，我之前写的姐妹文章里面提供了一个很好的delete操作符重载实现，如下代码所示。

```
void operator delete(void* var){
         Person *tmp=static_cast<Person*>(var);
        if(tmp->d_secur){
             std::cout<<"delete前对用户敏感信息重置为默认值"<<std::endl;
             tmp->d_name="";
             tmp->d_age=0;
             tmp->d_height=0;
             tmp->d_inNo="";
             tmp->show();
        }
        ::operator delete (var);
        var=nullptr;
 }
```

仅当Person对象内部的d_secu为true时,d_secure是一个安全标记，用于在new操作符初始化一个Person对象时,并且在该Person对象生命周期完结时，告知delete操作符如果存在敏感数据,就必须在回收前的堆内存块中的Person对象数据重置为0或其他默认值.

好了，这是我内存泄漏话题的第一篇，就目前简述而言,相比所有类似文章，我相信已经讲的很深入浅出的了。如果你觉得我的文章对你有所帮助的，可以关注我，并且分享给你的其他圈子，但请注明出处。

编辑于 2020-11-06 05:49

C++ 内存管理 string

C/C++内存管理
侧重C/C++内存模型，反编译分析

推荐阅读

C++:为何不建议用string作为函数参数

黄裕玲 发表于C语言程序...

C++ string类（学习笔记：第6章 23）

string类[1]使用字符串类string表示字符串（string类是C++标准库中一个封装起来的字符串）string实际上是对字符数组操作的封装（string类是C++标准库中一个封装起来的字符数组）...

品颜完月 发表于开发积累

C++ 类在内存中的存(一)

说了这么久的 C++ 终了，还是从内存出发来C++ 的类在内存中的存(之前写过一篇内存对齐类同样在一定程度上遵齐原则，不过比结构体

可乐船长2... 发表

还没有评论

写下你的评论...