## 68 Virtual Destructors in C++

前情提要: 26 Destructors in C++, 27 Inheritance in C++.28 Virtual Functions in C++

虚析构函数可以想象为虚函数和析构函数的组合。

虚析构函数对于处理多态非常重要,换句话说,如果我有一系列的子类和所有的继承:有一个类A,然后一个类B派生于A,你想把类B引用为类A,但它实际上是类B,然后你决定删除A或者它以某种方式删除了,然后你还是希望运行B的析构函数,而不是运行A的析构函数,这就是所谓的虚析构函数以及它的作用。

简单理解下就直接看代码吧:

```
#include <iostream>
class Base
public:
    Base() { std::cout << "Base Constructor\n"; }</pre>
    ~Base() { std::cout << "Base Destructor\n"; }
};
class Derived : public Base
public:
    Derived() { std::cout << "Derived Constructor\n"; }</pre>
    ~Derived() { std::cout << "Derived Destructor\n"; }
};
int main()
    Base* base = new Base();
    delete base;
    std::cout << "----\n";
    Derived* derived = new Derived();
    delete derived;
   std::cin.get();
}
```

很可能已经猜到运行结果了:

创建和删除Base类时只会调用Base类的构造和析构函数;

对于Derived类,首先调用了基类的构造函数,然后是Derived类的构造函数,当我们删除时先调用Derived类的析构函数,再调用基类Base类的析构函数。

```
delete base;
std::cin.get():
C:\Dev\HelloWorld\bin\Win

Base Constructor
Base Destructor
Base Constructor
Destructor
Derived Constructor
Derived Destructor
Destructor
Destructor
Destructor
Base Destructor
Destructor
Destructor
Base Destructor
Base
```

## 新加入一个多态类型:

```
std::cout << "-----\n";
Base* poly = new Derived(); // 创建一个Derived实例, 但是把它赋值给Base类
delete poly;
```

Base Destructor -----Base Constructor Derived Constructor Base Destructor

这里只有基类的析构函数被调用了,而派生类的析构函数没有被调用。

这点很重要,因为这会造成内存泄漏。

delete poly时,它不知道这个调用的析构函数可能有另一个析构函数,因为它(~Base)没有被标记为虚函数。

标记为virtual, 意味着C++知道在层次结构下可能有某种重写的方法,这个方法就可以被覆写。

而virtual destructor(虚析构函数)的意思不是覆写析构函数,而是**加上一个**析构函数。换句话说如果我把积累的析构函数改为虚函数,它实际会先调用派生类析构函数,然后在层次结构中向上,调用基类析构函数。

举一个例子来说明为什么这会造成内存泄漏:

```
// 给派生类加一个数组成员
class Derived: public Base
{
public:
    Derived() { m_Array = new int[5]; std::cout << "Derived Constructor\n"; }
    ~Derived() { delete[] m_Array; std::cout << "Derived Destructor\n"; }
private:
    int* m_Array;
};
```

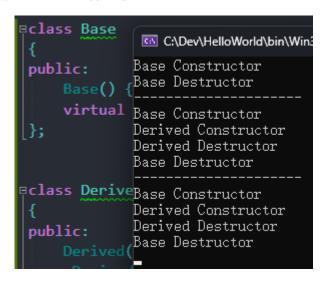
再回想一下刚才运行程序的结果,很明显这里并没有调用派生类的析构函数,因此给数组分配的20字节内存没有被释放,造成了内存泄漏。

## 那如何解决呢?

很简单,给基类的析构函数加上virtual即可:

```
class Base
{
public:
    Base() { std::cout << "Base Constructor\n"; }
    virtual ~Base() { std::cout << "Base Destructor\n"; }
};</pre>
```

这意味着这个类有可能拓展出子类,可能还有一个析构函数也需要被调用:



很好,现在有了和第二个例子完全一样的结果,意味着即使我们把它当做多态类型处理(或者说是当做基类类型处理),基类和派生类的析构函数都调用了,那个数组也得到了清理。

所以当你在写一个要拓展的类或者子类时,只要你允许一个类拥有子类,你100%需要声明你的析构函数为虚函数,否则无法安全地拓展这个类。