2020/05/11_C++_第5课_内存结构、this指针、构造析构、new、delete

笔记本: C++
创建时间: 2020/5/11 星期一 15:34
作者: ileemi
标签: new和delete, this指针, 构造析构, 结构、类的内存结构

- 内存结构
- this指针
- 构造函数、析构函数
- new和delete

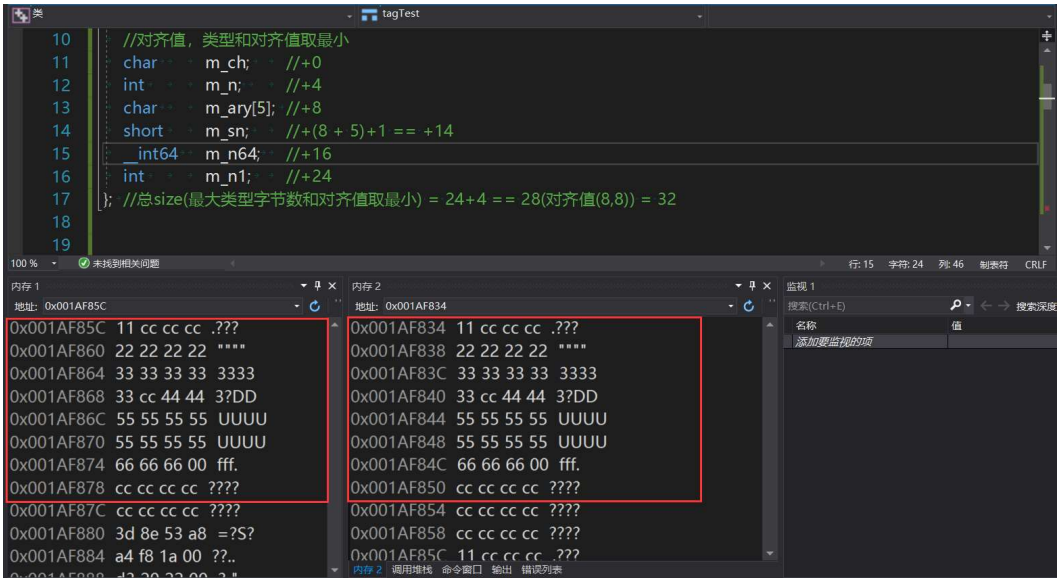
内存结构

C++中类的成员内存布局和C结构体中的成员内存布局规则一致（学习多态之前）
示例代码：

```
#include <iostream>
using namespace std;

//默认对齐值为8字节
struct tagTest
{
    //对齐值，类型和对齐值取最小
    char m_ch;  //+0
    int  m_n;   //+4
    char m_ary[5];  //+8
    short    m_sn;  //+(8 + 5)+1 == +14
    __int64  m_n64; //+16
    int  m_n1;  //+24
};
//总size(最大类型字节数和对齐值取最小) = 24+4 == 28(对齐值(8,8)) = 32

class CTest
{
public:
    char m_ch;
    int  m_n;
    char m_ary[5];
    short    m_sn;
    __int64  m_n64;
    int  m_n1;
```

```
};

int main()
{

    tagTest tt;
    CTest ct;

    //分别显示结构体和类的字节大小
    cout << sizeof(tt) << endl;
    cout << sizeof(ct) << endl;

    //为结构体成员赋值
    tt.m_ch = 0x11;
    tt.m_n = 0x22222222;
    memset(tt.m_ary, 0x33, sizeof(tt.m_ary));
    tt.m_sn = 0x4444;
    tt.m_n64 = 0x5555555555555555;
    tt.m_n1 = 0x666666;

    //为类中的数据成员赋值
    ct.m_ch = 0x11;
    ct.m_n = 0x22222222;
    memset(ct.m_ary, 0x33, sizeof(ct.m_ary));
    ct.m_sn = 0x4444;
    ct.m_n64 = 0x5555555555555555;
    ct.m_n1 = 0x666666;

    return 0;
}
```

内存结构如下：



this指针

this 是 C++ 中的一个关键字，也是一个 const 指针，它指向当前对象，通过它可以访问当前对象的所有成员。

this 只能用在类的内部，通过 this 可以访问类的所有成员，包括 private、protected、public 属性的。

注意，this 是一个指针，要用->来访问成员变量或成员函数。
当类中的成员函数的参数和成员变量重名，只能通过 this 指针进行区分

this 虽然用在类的内部，但是只有在对象被创建以后才会给 this 赋值，并且这个赋值的过程是编译器自动完成的，不需要用户干预，用户也不能显式地给 this 赋值，this 指针的值和 类对象指针的值是相同的。

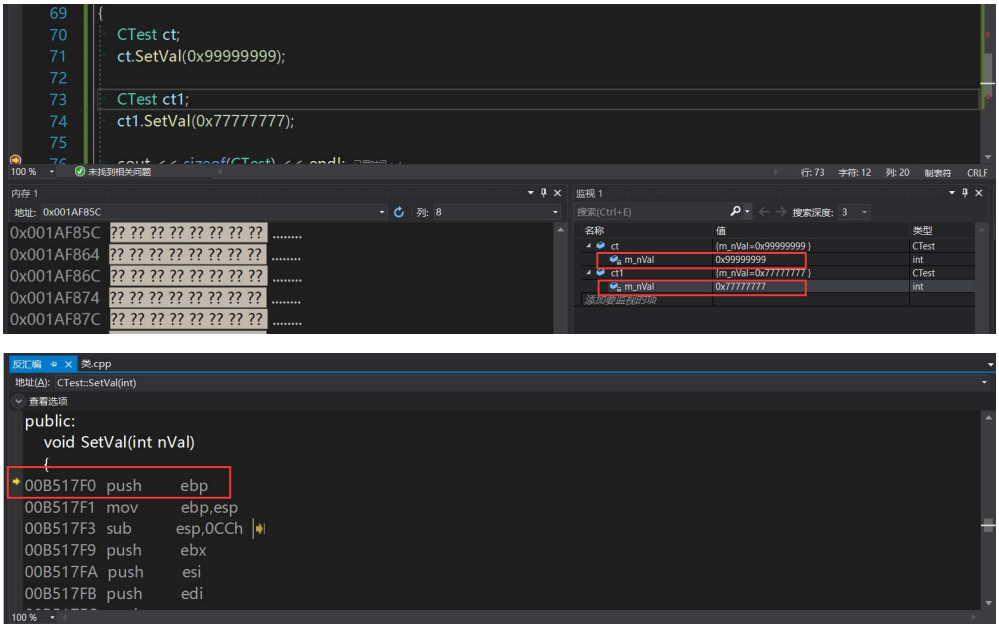
注意：

- this 是 const 指针，它的值是不能被修改的，一切企图修改该指针的操作，如赋值、递增、递减等都是不允许的。
- this 只能在成员函数内部使用，用在其他地方没有意义，也是非法的。
- 只有当对象被创建后 this 才有意义，因此不能在 static 成员函数中使用（后续会讲到 static 成员）

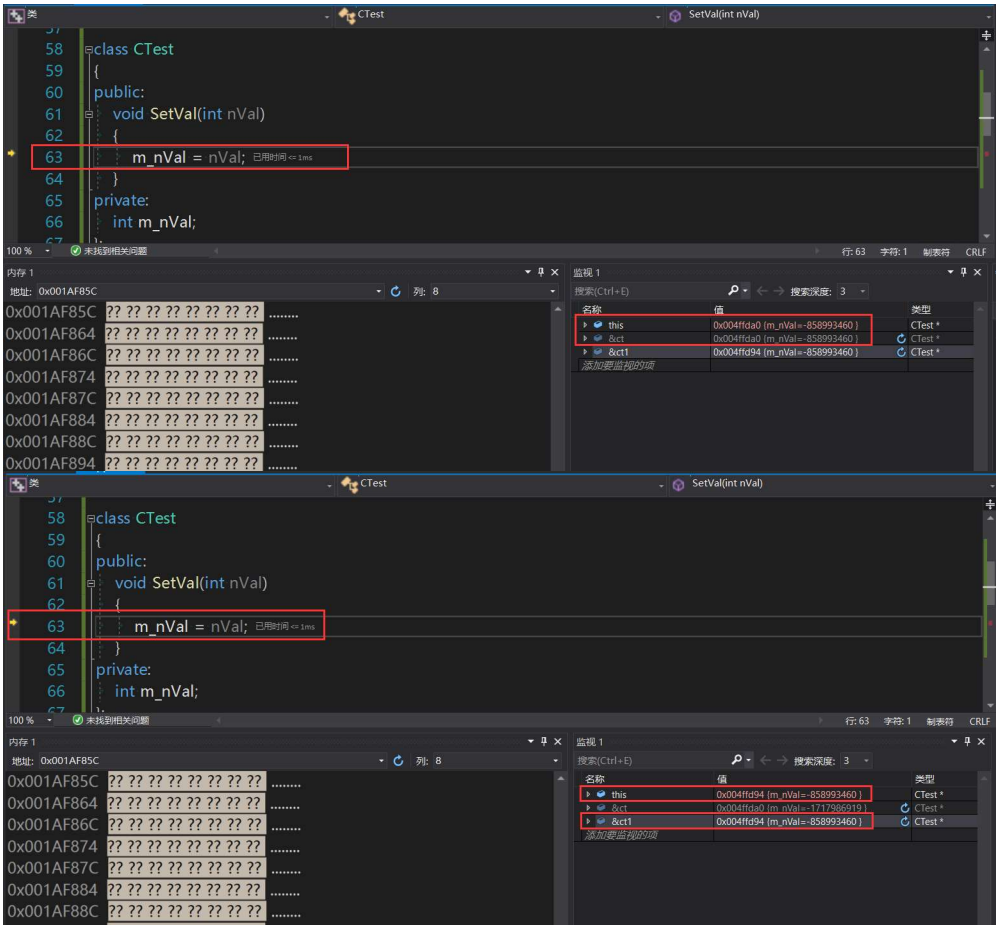
this 实际上是成员函数的一个形参，在调用成员函数时将对象的地址作为实参传递给 this。不过 this 这个形参是隐式的，它并不出现在代码中，而是在编译阶段由编译器默默地将它添加到参数列表中。

this 作为隐式形参，本质上是成员函数的局部变量，所以只能用在成员函数的内部，并且只有在通过对象调用成员函数时才给 this 赋值。

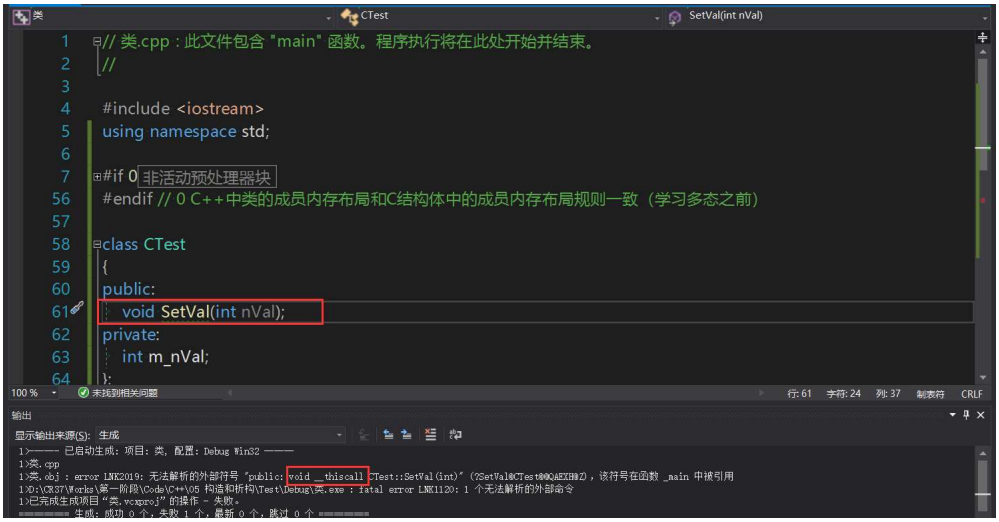
- 对于不同的对象，数据成员是独有的，成员函数是共享的，内存中访问的地址相同



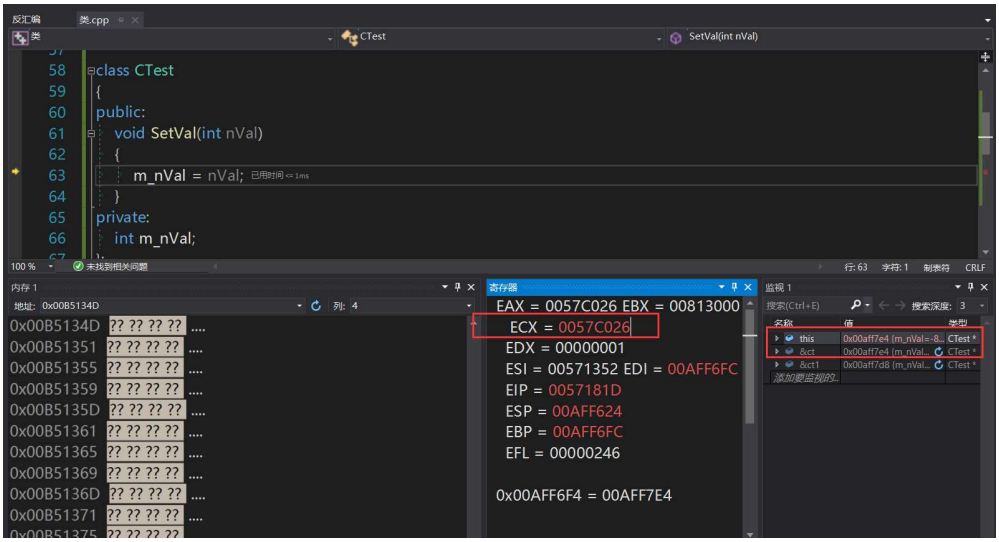
- 编译调用成员函数的时候,会默认传入this指针, this指针指向了调用函数的对象的首地址

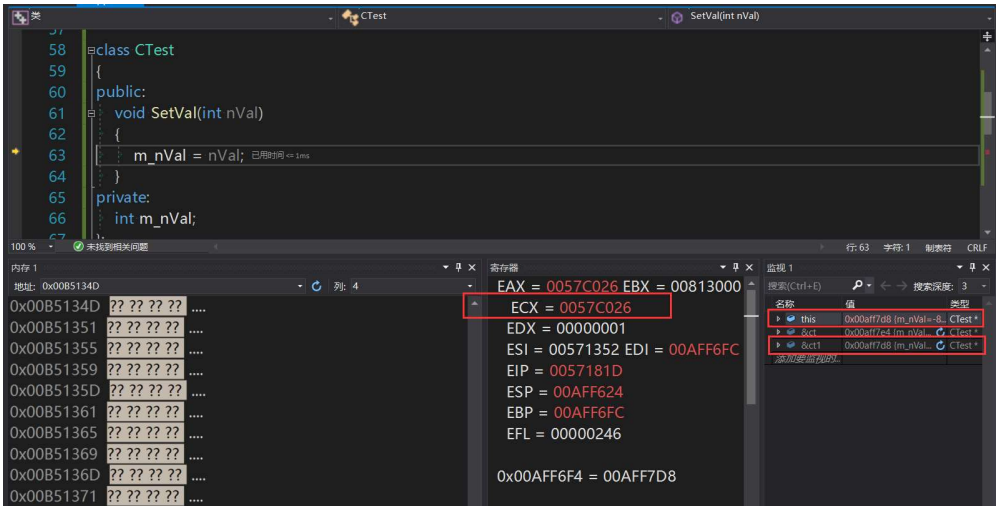


这种调用约定是__thiscall(只能用在成员函数上)，不是关键字

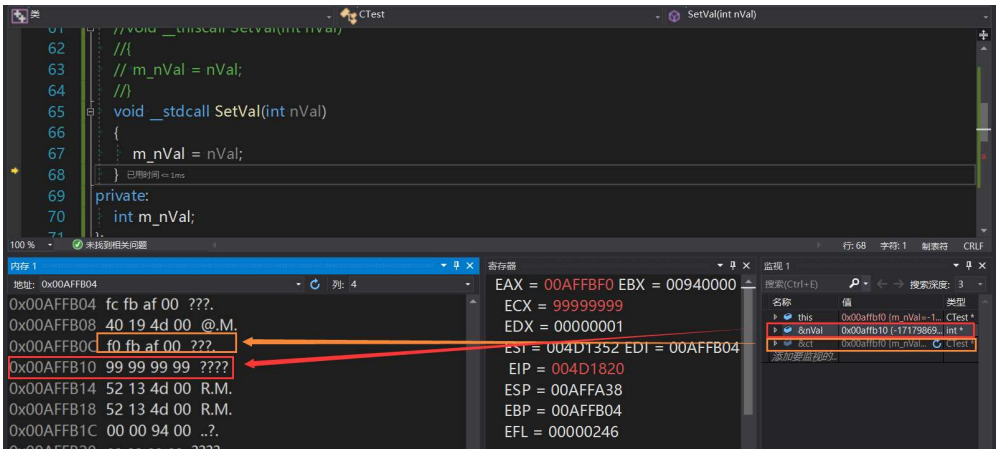


- this指针通过寄存器ECX传入(栈可以传递参数外，寄存器也可以传递参数)





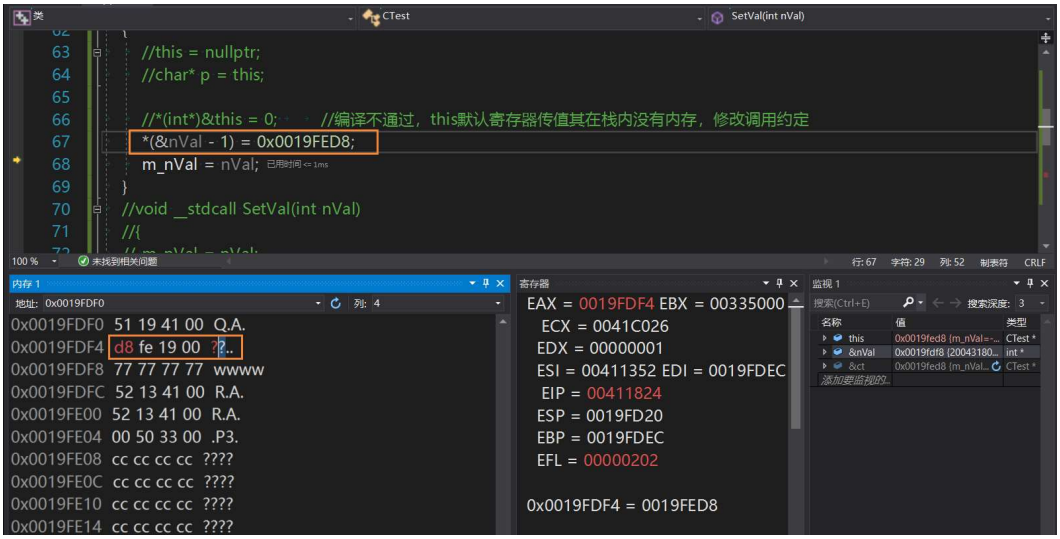
- 成员函数的调用约定可以修改, 但是不建议修改, 了解即可



- *this的类型是className const this，this本身不能被修改，其指向的成员可以被修改（指针本身不可以修改，指针指向的内容可以被修改）*

  语法层面不可以修改：可使用下面的技巧修改this本身

```
void __cdecl SetVal(int nVal)
{
    *(&nVal - 1) = 0x0019FED8;
    m_nVal = nVal;
}
```



构造函数、析构函数

手动调用初始化函数的缺陷：

- 忘记调用初始化函数
- 多次调用初始化函数，会造成内存泄漏（调用一次申请一次内存，没有机会进行释放）

手动调用反初始化函数的缺陷：

- 忘记调用反初始化函数
- 忘记调用初始化函数

为了解决手动调用初始化函数和反初始化函数，C++提供了构造函数和析构函数，按照规定的方式，编译器自动进行调用。

调用构造函数和析构函数可以手动调用，但是不建议但是强烈不建议手动调用，了解即可
示例：

```
class CTest
{
public:
    CTest();
    ~CTest();
};
int main()
{
    CTest ctest;                //实例化一个类对象

    ctest.CTest::CTest();       //手动调用构造函数
    return 0;
}
```

构造函数(初始化函数)：

- 函数名和类名相同，没有返回值，可以有参数，可以带默认参，**构造函数可以重载**

析构函数(反初始化函数)：

- 函数名和类名相同，前面加~，无参，无返回值，析构函数不能重载(同时也没有重载的必要)

构造函数和析构函数的调用时机：

受作用域影响

- 构造在对象的生命周期的开始时被调用
- 析构在对象的生命周期的结束时被调用

在块作用域内示例化一个对象，在对象创建时调用构造函数，在块作用域结束时，调用析构函数

构造函数的花式调用方式(了解即可):

```
class CTest
{
public:
    CTest(const char* pData = nullptr, int nCount = 0);      //1
    CTest(int nVal);    //2
int main()
{
    CTest ctest;    //1

    CTest ctest1("Hello", 6);   //1
    CTest buf2 = 999;   //2
    CTest buf3 = { 100 };   //2
    CTest buf4 = { "Test", 5 }; //1
    CTest buf5{ "Haha", 3 }; //1
}
```

new和delete

在堆里实例化一个对象使用C的malloc，需要下面的操作

```
class CTest
{
public:
   CTest(const char* pData = nullptr, int nCount = 0)
   {
       cout << "CTest::CTest" << endl;
       m_nCount = nCount;
   }

   CTest(int nVal)
   {
       cout << "CTest::CTest" << endl;
       m_nCount = nVal;
   }

   ~CTest()
   {
       cout << "CTest::~CTest" << endl;
   }
private:
   int m_nCount;
};
```

```
int main()
{
   CTest* pTest = (CTest*)malloc(sizeof(CTest));
   pTest->CTest::CTest();      //同样需要手动调用构造函数进行初始化
   pTest->~CTest();
   free(pTest);
```

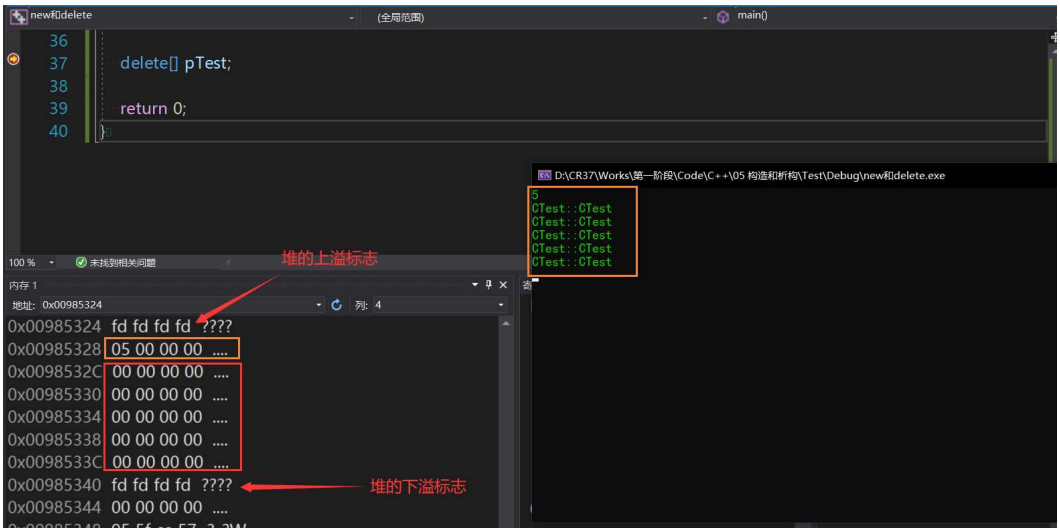
C++中使用 new和delete

```
int main()
{

    //申请堆内存，调用构造
    CTest* pTest = new CTest(0x87654093);
    //调用析构，释放堆内存
    delete pTest;


    //使用new申请一个数组
    CTest* pTestAry = new CTest[5]; //每个对象都会调用一次构造
    delete[] pTestAry;  //调5次析构

}
```

申请堆数组时，编译器会会根据申请的次数调用对应的构造和析构

```
#include <iostream>
using namespace std;

class CTest
{
public:
        CTest(const char* pData = nullptr, int nCount = 0)
        {
                cout << "CTest::CTest" << endl;
                m_nCount = nCount;
        }
        CTest(int nVal)
        {
                cout << "CTest::CTest" << endl;
                m_nCount = nVal;
        }
        ~CTest()
        {
                cout << "CTest::~CTest" << endl;
```

```
        }
private:
        int m_nCount;
};

int main()
{
        int nCount = 0;
        cin >> nCount;
    //编译器根据输入的nCount值调用对应次数的构造
        CTest* pTest = new CTest[nCount];

    //编译器根据输入的nCount值调用对应次数和析构
        delete[] pTest;


   //delete pTest; //delete类型不匹配，释放失败，程序崩溃

   //delete (CBuff*)((int*)pBufAry0 - 1); //手动减int字节，只调用一次析构


        return 0;
}
```



同时还可以手动修改析构的次数

```
int main()
{
        int nCount = 0;
        cin >> nCount;
    //编译器根据输入的nCount值调用对应次数的构造
        CTest* pTest = new CTest[nCount];
```

```
    (*((int*)pTest-1))--; //手动将调用析构次数减1

    delete[] pTest;

    return 0;
}
```