2020/05/14_C++_第8课_友元函数、引用计数

笔记本:	C++
创建时间:	2020/5/14 星期四 15:32
作者:	ileemi
标签:	引用计数, 友元函数

友元函数

介绍：类的友元函数是定义在类外部，但有权访问类的所有私有（private）成员和保护（protected）成员。尽管友元函数的原型有在类的定义中出现过，但是友元函数并不是成员函数。

友元可以是一个函数，该函数被称为友元函数；友元也可以是一个类，该类被称为友元类，在这种情况下，整个类及其所有成员都是友元。如果要声明函数为一个类的友元，需要在类定义中该函数原型前使用关键字 friend，例如：

```
//友元函数
friend int main();

//友元类
friend class CTest;
```

友元函数（缺点：破坏封装性）：
1、可以访问类的所有的成员(公有的和私有的), 不受访问标号的影响
2、可以声明友元函数, 友元成员函数, 友元类
3、友元函数可以定义在类外面, 也可以定义在类里(仍然是全局函数, 但是需要在类外面声明)

示例：
1、可以访问类的所有的成员(公有的和私有的), 不受访问标号的影响

```
将main函数设置为该的类的友元函数
//main函数内部直接访问类的数据成员进行赋值操作
#include <iostream>
using namespace std;

//友元
class CFoo
{
```

```
public:
    void SetVal(int nVal)
    {
        m_nVal = nVal;
    }
    int GetVal()const
    {
        return m_nVal;
    }
     //友元函数
    friend int main();
    //可在main函数内部直接访问类的数据成员进行赋值操作
private:
    int m_nVal;
};
int main()
{
                //友元函数，可通过对象直接访问类中的数据成员
                CFoo foo;
                foo.m_nVal = 666;
                return 0;
}
```

2. 可以声明友元函数, 友元成员函数, 友元类
3. 友元函数可以定义在类外面, 也可以定义在类里(仍然是全局函数, 但是需要在类外面声明)

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

class CTest
{
public:
    void Test1();
    void Test2();
    void Test3();
    void Test4();
};

class CFoo
{
public:
    void SetVal(int nVal)
    {
        m_nVal = nVal;
    }
    int GetVal()const
```

```
    {
        return m_nVal;
    }

    //友元函数

    friend int main();                          //可在main函数内部直接访问类
的数据成员进行赋值操作
    friend void Test()
    {
        CFoo foo;
        foo.m_nVal = 777;
    }

    //友元成员函数
    friend void CTest::Test1();
    //在没有在该类中声明友元类的前提下，此时CTest类中的成员函数Test1可以
访问该类的数据成员
    friend void CTest::Test2();
    //在没有在该类中声明友元类的前提下，此时CTest类中的成员函数Test1可以
访问该类的数据成员

    //友元类
    friend class CTest;
    //添加友元类后，添加的类成员函数可以访问该类中的数据成员，并可以修改
其值
private:
    int m_nVal;
};

void CTest::Test1()
{
    CFoo foo;
    foo.m_nVal = 999;
}

void CTest::Test2()
{
    CFoo foo;
    foo.m_nVal = 999;
}
void CTest::Test3()
{

    CFoo foo;
    /*
    在没有在该类中声明友元类的前提下，
```

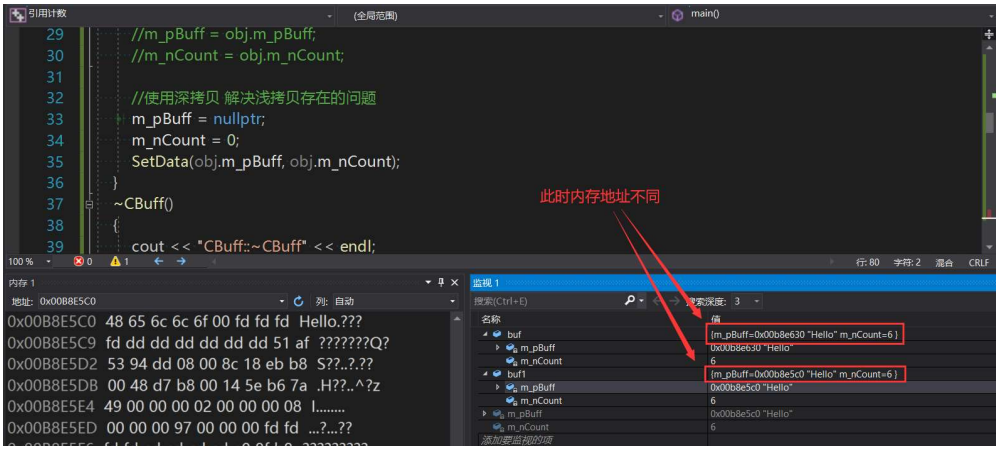
```
    该成员函数不是CFoo类中的友元成员函数，不能访问该类中的数据成员
    */
    foo.m_nVal = 999;
}

void CTest::Test4()
{
    CFoo foo;
    /*
    在没有在该类中声明友元类的前提下，

    该成员函数不是CFoo类中的友元成员函数，不能访问该类中的数据成员

    */
    foo.m_nVal = 999;
}

//友元函数可以定义在类内，其仍是全局函数，但是需要在类外进行声明，才能使用
void Test();

int main()
{
    //友元函数
    CFoo foo;
    foo.SetVal(999);
    cout << foo.GetVal() << endl;

    //将main函数设置为该类的友元函数，可直接通过对象修改类中的数据成员数值
    foo.m_nVal = 666;

    //定义在类内的友元函数，仍然是全局函数，但是需要在类外面声明
    CFoo footest;
    //CFoo::Test();
    Test();

    //友元成员函数
    CTest test;
    test.Test1();
    test.Test2();
    return 0;
}
```

引用计数

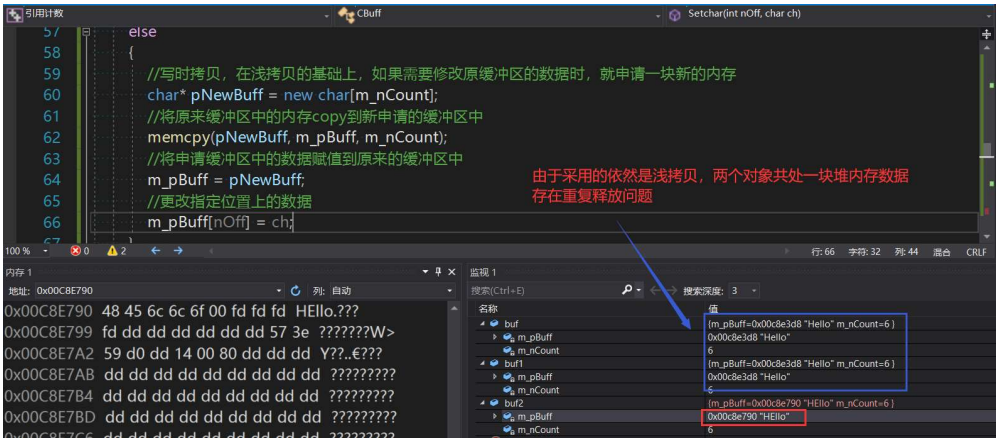
1、浅拷贝

- 优点：省内存, 效率高
- 缺点：存在内存的重复释放问题，所有对象使用同一块内存数据，修改一个对象，将影响所有对象，重复释放(重复调用析构，程序崩溃)
- 

2、深拷贝

- 优点：不会重复释放
- 缺点：和浅拷贝存在相同的问题，当每次修改对象的时候，会重新申请新的内存空间，存放在不同的堆空间，会造成内存浪费，效率低下
- 

3、写时拷贝（浅拷贝和深拷贝混合）

- 优点：当某个对象做修改的时候, 给它分配一块单独的内存, 修改在新的内存中修改
- 缺点：依然存在重复释放内存的问题
- 

- 

4、引用计数（解决写时拷贝中的浅拷贝问题）

可通过添加一个计数器(用于解决上述的重复释放内存的问题)，对存在的对象进行计数，当对象的对象销毁的受，计数减1，当计数器为0的时候就进行销毁堆空间。

分析计数器应该放在哪里呢?

1）放在类里做成员

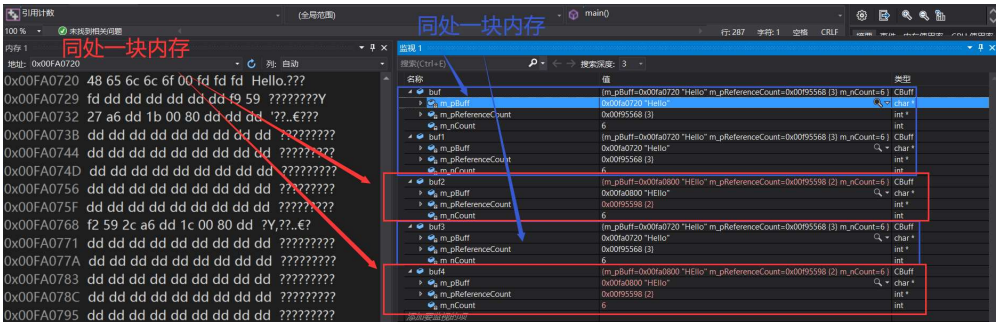
- 存在问题：当每个对象销毁的时候，需要通知其他的对象将计数器进行减一，但是做不到。

2）设置为静态成员

- 存在问题：写时拷贝会申请新的内存，新的内存也需要进行引用计数，当新内存过多，就无法估算有多少个新内存，所以无法确定静态成员的数量。

3）随对象一起和数据一起放置到堆空间里

- 解决办法：将计数器放置到堆中，每块堆分配一个计数器



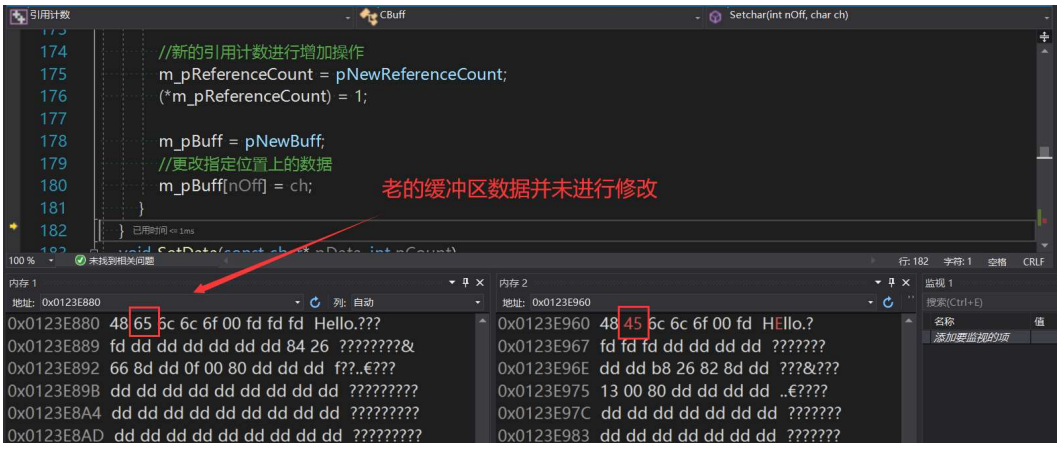
代码示例：

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

//引用计数
class CBuff
{
public:
	//默认构造初始化类数据
CBuff(const char* pData = nullptr, int nCount = 0)
{
    cout << "CBuff::CBuff" << endl;
    if (pData == nullptr)
    {
        m_pBuff = nullptr;
        m_pReferenceCount = 0; //初始化引用计数的数值
        m_nCount = 0;
    }
    else
    {
        //申请新的内存，存入新的数据
        AllocNewBuff(pData, nCount);
    }
}
//拷贝构造
CBuff(const CBuff& obj)
```

```
{
    //和默认构造的功能一样
    //浅拷贝(存在重复释放)
    m_pBuff = obj.m_pBuff;
    m_nCount = obj.m_nCount;
    m_pReferenceCount = obj.m_pReferenceCount;
    //引用计数++操作
    (*m_pReferenceCount)++;

    //使用深拷贝 解决浅拷贝存在的问题
    //m_pBuff = nullptr;
    //m_nCount = 0;
    //SetData(obj.m_pBuff, obj.m_nCount);

}
~CBuff()
{
    cout << "CBuff::~CBuff" << endl;
    ReleaseBuff();
}

//从缓冲区中将下标为nOff的位置，替换其字符串为ch
void Setchar(int nOff, char ch)
{
    if (nOff > m_nCount)
    {
        return;
    }
    else
    {
        //写时拷贝，在浅拷贝的基础上，如果需要修改原缓冲区的数据时，就申请一块新的内存
        char* pNewBuff = new char[m_nCount];

        //修改缓冲区中的数据时，需要申请新的缓冲区，所以同时也申请新的引用计数器
        int* pNewReferenceCount = new int;

        //将原来缓冲区中的内存copy到新申请的缓冲区中
        memcpy(pNewBuff, m_pBuff, m_nCount);

        //将申请缓冲区中的数据赋值到原来的缓冲区中
        //此时原来的引用计数需要进行减减操作
        (*m_pReferenceCount)--;

        //新的引用计数进行增加操作
        m_pReferenceCount = pNewReferenceCount;
```

```
            (*m_pReferenceCount) = 1;
            m_pBuff = pNewBuff;

            //更改指定位置上的数据
            m_pBuff[nOff] = ch;
        }
    }
    void SetData(const char* pData, int nCount)
    {
        //释放之前的内存
        ReleaseBuff();
        //申请新的内存，存入新的数据
        AllocNewBuff(pData, nCount);
    }
    const char* GetData()
    {
        return m_pBuff;
    }

    private:
    //释放内存
    void ReleaseBuff()
    {
        //调用析构，释放堆内存之前，应判断引用计数是否为0，为0释放
        --(*m_pReferenceCount);          //调用一次析构，引用计数进行减1操作

        //释放之前的内存
        if (m_pBuff != nullptr)
        {
            //当引用计数的值为0的时候，说明没有其它对象持有这块内存，因此，
    可以释放这块内存
            if ((*m_pReferenceCount) == 0)
            {
                delete[] m_pBuff;
                delete m_pReferenceCount;
                m_pBuff = nullptr;
                m_pReferenceCount = nullptr;
                m_nCount = 0;
            }
        }
    }
    void AllocNewBuff(const char* pData, int nCount)
    {
        m_nCount = nCount;
        //申请新的内存，存入新的数据
        m_pBuff = new char[m_nCount];
        //申请存储计数器的堆内存
```

```
        m_pReferenceCount = new int;
        memcpy(m_pBuff, pData, m_nCount);
        //计数器数值置1
        *m_pReferenceCount = 1;
    }
    private:
            char* m_pBuff;
            int* m_pReferenceCount;  //引用计数(每块堆申请一个引用计数)
            int m_nCount; //字符数据的长度
    };

    int main()
    {

        /*
            CBuff buf("Hello", 6);
            CBuff buf1(buf);                    //调用拷贝构造
            CBuff buf2(buf1);

            buf2.Setchar(1, 'E');          //依然存在重复释放问题

            CBuff buf3(buf1);
            CBuff buf4(buf2);
        */
        CBuff buf("Hello", 6);
        //此处存在一个Bug，修改数据的时候，原来的缓冲区数据并未进行修改
        buf.Setchar(1, 'E');

            return 0;
    }
```



需要在Setchar()方法下做如下修改：

```
    void Setchar(int nOff, char ch)
    {
        if (nOff > m_nCount)
        {
```

```
            return;
        }
        else
        {
            //解决上述的Bug这里就需要新建一个临时的缓冲区
            CBuff buf(m_pBuff, m_nCount);

            //释放缓冲区
            ReleaseBuff();

            //重新申请一块缓冲区
            AllocNewBuff(buf.m_pBuff, buf.m_nCount);

            //修改对应的数值
            m_pBuff[nOff] = ch;
        }
    }
```