C++ 引用计数技术及智能指针的简单实现

一直以来都对智能指针一知半解，看C++Primer中也讲的不够清晰明白（大概是我功力不够吧）。最近花了点时间认真看了智能指针，特地来写这篇文章。

1.智能指针是什么

简单来说，智能指针是一个类，它对普通指针进行封装，使智能指针类对象具有普通指针类型一样的操作。具体而言，复制对象时，副本和原对象都指向同一存储区域，如果通过一个副本改变其所指的值，则通过另一对象访问的值也会改变.所不同的是，智能指针能够对内存进行进行自动管理，避免出现悬垂指针等情况。

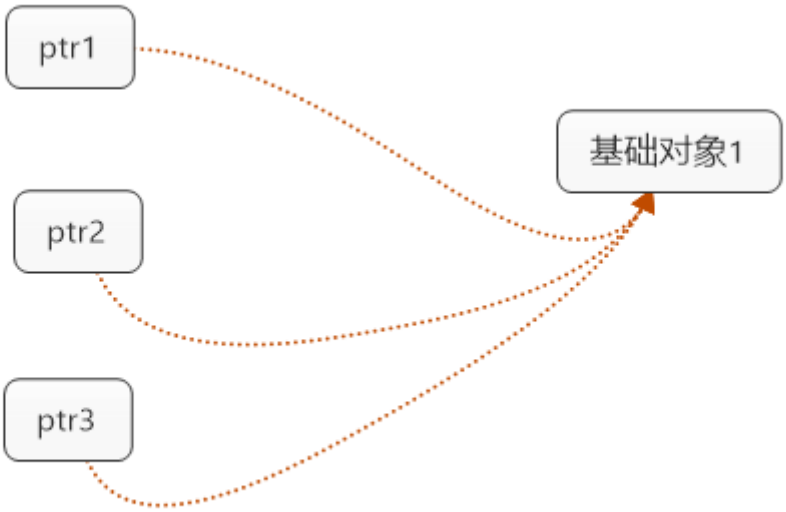
2.普通指针存在的问题

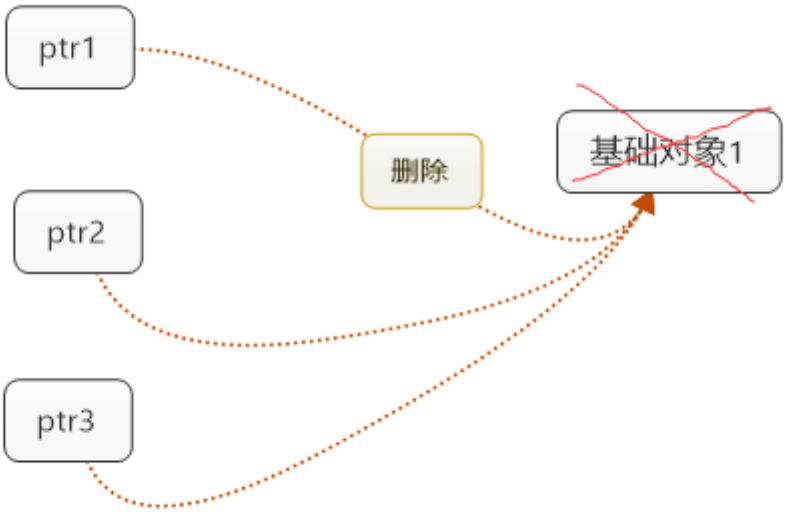
C语言、C++语言没有自动内存回收机制，关于内存的操作的安全性依赖于程序员的自觉。程序员每次new出来的内存块都需要自己使用delete进行释放，流程复杂可能会导致忘记释放内存而造成内存泄漏。而智能指针也致力于解决这种问题，使程序员专注于指针的使用而把内存管理交给智能指针。

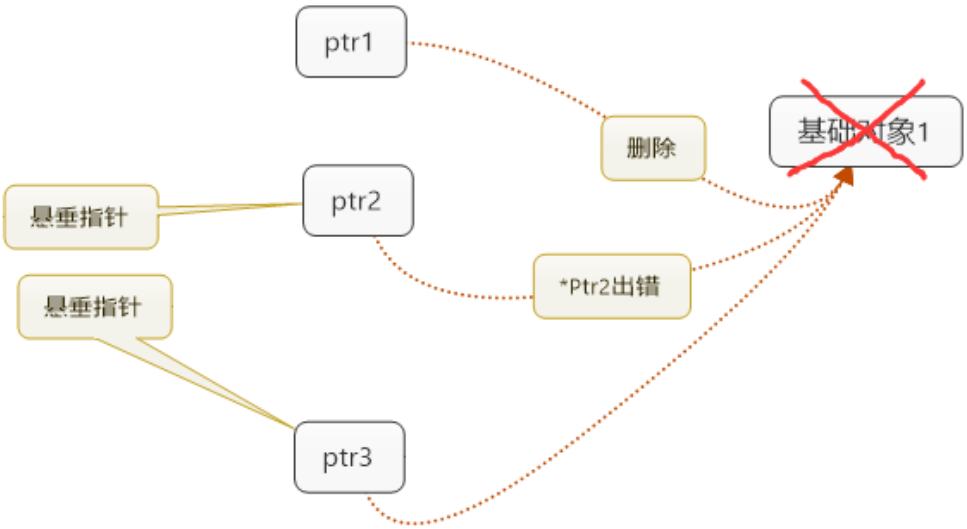
我们先来看看普通指针的悬垂指针问题。当有多个指针指向同一个基础对象时，如果某个指针delete了该基础对象，对这个指针来说它是明确了它所指的对象被释放掉了，所以它不会再对所指对象进行操作，但是对于剩下的其他指针来说呢？它们还傻傻地指向已经被删除的基础对象并随时准备对它进行操作。于是悬垂指针就形成了，程序崩溃也“指日可待”。我们通过代码+图来来探求悬垂指针的解决方法。

```
        int * ptr1 = new int (1);
        int * ptr2 = ptr1;
        int * ptr3 = prt2;

        cout << *ptr1 << endl;
        cout << *ptr2 << endl;
        cout << *ptr3 << endl;

        delete ptr1;

        cout << *ptr2 << endl;
```

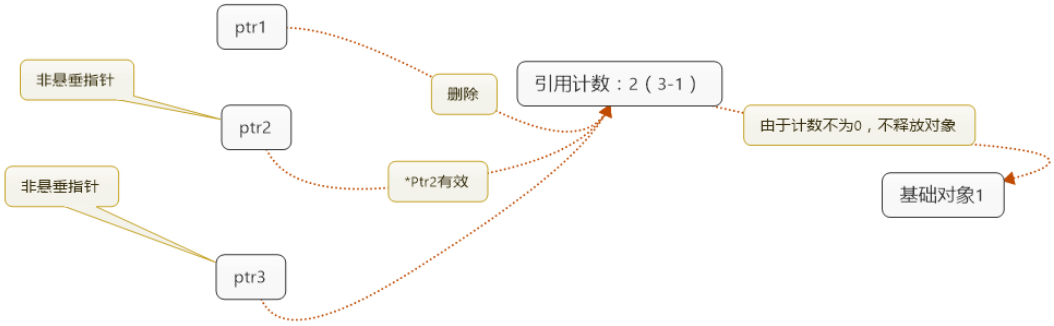
代码简单就不啰嗦解释了。运行结果是输出**ptr2**时并不是期待的1，因为1已经被删除了。这个过程是这样的：







从图可以看出，错误的产生来自于**ptr1**的"无知"：它并不知道还有其他指针共享着它指向的对象。如果有个办法让**ptr1**知道，除了它自己外还有两个指针指向基础对象，而它不应该删除基础对象，那么悬垂指针的问题就得以解决了。如下图：



那么何时才可以删除基础对象呢？当然是只有一个指针指向基础对象的时候，这时通过该指针就可以大大方方地把基础对象删除了。

3.什么是引用计数

如何来让指针知道还有其他指针的存在呢？这个时候我们该引入**引用计数**的概念了。引用计数是这样一个技巧，它允许有多个相同值的对象共享这个值的实现。引用计数的使用常有两个目的：

- 简化跟踪堆中（也即C++中new出来的）的对象的过程。一旦一个对象通过调用new被分配出来，记录谁拥有这个对象是很重要的，因为其所有者要负责对它进行delete。但是对象所有者可以有多个，且所有权能够被传递，这就使得内存跟踪变得困难。引用计数可以跟踪对象所有权，并能够自动销毁对象。可以说引用计数是个简单的垃圾回收体系。这也是本文的讨论重点。
- 节省内存，提高程序运行效率。如何很多对象有相同的值，为这多个相同的值存储多个副本是很浪费空间的，所以最好做法是让左右对象都共享同一个值的实现。C++标准库中**string**类采取一种称为"写时复制"的技术，使得只有当字符串被修改的时候才创建各自的拷贝，否则可能（标准库允许使用但没强制要求）采用引用计数技术来管理共享对象的多个对象。这不是本文的讨论范围。

4.智能指针实现

了解了引用计数，我们可以使用它来写我们的智能指针类了。智能指针的实现策略有两种：辅助类与句柄类。这里介绍辅助类的实现方法。

4.1.基础对象类

首先，我们来定义一个基础对象类Point类，为了方便后面我们验证智能指针是否有效，我们为Point类创建如下接口：

```
class Point
{
public:
    Point(int xVal = 0, int yVal = 0) :x(xVal), y(yVal) { }
    int getX() const { return x; }
    int getY() const { return y; }
    void setX(int xVal) { x = xVal; }
    void setY(int yVal) { y = yVal; }

private:
    int x, y;

};
```

4.2.辅助类

在创建智能指针类之前，我们先创建一个辅助类。这个类的所有成员皆为私有类型，因为它不被普通用户所使用。为了只为智能指针使用，还需要把智能指针类声明为辅助类的友元。这个辅助类含有两个数据成员：计数**count**与基础对象指针。也即**辅助类用以封装使用计数与基础对象指针。**

```
class U_Ptr
{
private:

    friend class SmartPtr;
    U_Ptr(Point *ptr) :p(ptr), count(1) { }
    ~U_Ptr() { delete p; }

    int count;
    Point *p;
};
```

4.3.为基础对象类实现智能指针类

引用计数是实现智能指针的一种通用方法。智能指针将一个计数器与类指向的对象相关联，引用计数跟踪共有多少个类对象共享同一指针。它的具体做法如下：

- 当创建类的新对象时，初始化指针，并将引用计数设置为1
- 当对象作为另一个对象的副本时，复制构造函数复制副本指针，并增加与指针相应的引用计数（加1）
- 使用赋值操作符对一个对象进行赋值时，处理复杂一点：先使左操作数的指针的引用计数减1（为何减1：因为指针已经指向别的地方），如果减1后引用计数为0，则释放指针所指对象内存。然后增加右操作数所指对象的引用计数（为何增加：因为此时做操作数指向对象即右操作数指向对象）。
- 析构函数：调用析构函数时，析构函数先使引用计数减1，如果减至0则delete对象。

做好前面的准备后，我们可以来为基础对象类Point书写一个智能指针类了。根据引用计数实现关键点，我们可以写出我们的智能指针类如下：

```
```

```
class SmartPtr
{
public:
    SmartPtr(Point *ptr) :rp(new U_Ptr(ptr)) { }

    SmartPtr(const SmartPtr &sp) :rp(sp.rp) { ++rp->count; }

    SmartPtr& operator=(const SmartPtr& rhs) {
        ++rhs.rp->count;
        if (--rp->count == 0)
            delete rp;
        rp = rhs.rp;
        return *this;
    }

    ~SmartPtr() {
        if (--rp->count == 0)
            delete rp;
        else
        cout << "还有" << rp->count << "个指针指向基础对象" << endl;
    }

private:
        U_Ptr *rp;
};
```

4.4.智能指针类的使用与测试

至此，我们的智能指针类就完成了，我们可以来看看如何使用

```
int main()
{
    //定义一个基础对象类指针
    Point *pa = new Point(10, 20);

    //定义三个智能指针类对象，对象都指向基础类对象pa
    //使用花括号控制三个指针指针的生命期，观察计数的变化

    {
        SmartPtr sptr1(pa);//此时计数count=1
        {
            SmartPtr sptr2(sptr1); //调用复制构造函数，此时计数为count=2
            {
                SmartPtr sptr3=sptr1; //调用赋值操作符，此时计数为conut=3
            }
            //此时count=2
        }
        //此时count=1；
    }
    //此时count=0；pa对象被delete掉

    cout << pa->getX ()<< endl;

    system("pause");
    return 0;
}
```

来看看运行结果咯：

```
还有2个指针指向基础对象
还有1个指针指向基础对象
-17891602
请按任意键继续. . .
```

如期，在离开大括号后，共享基础对象的指针从3->2->1->0变换，最后计数为0时，pa对象被delete，此时使用getX（）已经获取不到原来的值。

5.智能指针类的改进一

虽然我们的**SmartPtr**类称为智能指针，但它目前并不能像真正的指针那样有->、*等操作符，为了使它看起来更像一个指针，我们来为它重载这些操作符。代码如下所示：

```
{
public:
    SmartPtr(Point *ptr) :rp(new U_Ptr(ptr)) { }

    SmartPtr(const SmartPtr &sp) :rp(sp.rp) { ++rp->count; }

    SmartPtr& operator=(const SmartPtr& rhs) {
        ++rhs.rp->count;
        if (--rp->count == 0)
            delete rp;
        rp = rhs.rp;
        return *this;
    }

    ~SmartPtr() {
        if (--rp->count == 0)
            delete rp;
        else
        cout << "还有" << rp->count << "个指针指向基础对象" << endl;
    }


    Point & operator *()         //重载*操作符
    {
        return *(rp->p);
    }
    Point* operator ->()        //重载->操作符
    {
        return rp->p;
    }

private:
    U_Ptr *rp;
};
```

然后我们可以像指针般使用智能指针类

```
    Point *pa = new Point(10, 20);
    SmartPtr sptr1(pa);
    //像指针般使用
    cout<<sptr1->getX();

```

6.智能指针改进二

目前这个智能指针智能用于管理Point类的基础对象，如果此时定义了个矩阵的基础对象类，那不是还得重新写一个属于矩阵类的智能指针类吗？但是矩阵类的智能指针类设计思想和Point类一样啊，就不能借用吗？答案当然是能，那就是使用模板技术。为了使我们的智能指针适用于更多的基础对象类，我们有必要把智能指针类通过模板来实现。这里贴上上面的智能指针类的模板版：

```
    //模板类作为友元时要先有声明
    template <typename T>
    class SmartPtr;

    template <typename T>
    class U_Ptr     //辅助类
    {
    private:
        //该类成员访问权限全部为private，因为不想让用户直接使用该类
        friend class SmartPtr<T>;       //定义智能指针类为友元，因为智能指针类需要直接操纵辅助类

        //构造函数的参数为基础对象的指针
        U_Ptr(T *ptr) :p(ptr), count(1) { }

        //析构函数
        ~U_Ptr() { delete p; }
        //引用计数
        int count;

        //基础对象指针
        T *p;
    };

    template <typename T>
    class SmartPtr   //智能指针类
    {
    public:
        SmartPtr(T *ptr) :rp(new U_Ptr<T>(ptr)) { }       //构造函数
        SmartPtr(const SmartPtr<T> &sp) :rp(sp.rp) { ++rp->count; }  //复制构造函数
        SmartPtr& operator=(const SmartPtr<T>& rhs) {     //重载赋值操作符
            ++rhs.rp->count;      //首先将右操作数引用计数加1，
            if (--rp->count == 0)      //然后将引用计数减1，可以应对自赋值
                delete rp;
            rp = rhs.rp;
            return *this;
        }

        T & operator *()         //重载*操作符
        {
            return *(rp->p);
        }
```

```
        T* operator ->()        //重载->操作符
        {
            return rp->p;
        }


        ~SmartPtr() {         //析构函数
            if (--rp->count == 0)     //当引用计数减为0时，删除辅助类对象指针，从而删除基础对象
                delete rp;
            else
            cout << "还有" << rp->count << "个指针指向基础对象" << endl;
        }
    private:
        U_Ptr<T> *rp;  //辅助类对象指针
    };
```

好啦，现在我们能够使用这个智能指针类对象来共享其他类型的基础对象啦，比如int：

```
int main()
{
    int *i = new int(2);
    {
        SmartPtr<int> ptr1(i);
        {
            SmartPtr<int> ptr2(ptr1);
            {
                SmartPtr<int> ptr3 = ptr2;

                cout << *ptr1 << endl;
                *ptr1 = 20;
                cout << *ptr2 << endl;

            }
        }
    }
    system("pause");
    return 0;
}
```

运行结果如期所愿，SmartPtr类管理起int类型来了：

```
        2
        20
        还有2个指针指向基础对象
        还有1个指针指向基础对象
        请按任意键继续. . .
```