**shared\_ptr出现原因**

通过第一章的学习，我们知道不管是auto\_ptr合适scoped\_ptr都是存在缺陷的，于是我们必须想出一个方法既能很好的管理我们的内存，而且在使用的时候，可以多个指针指向一个内存，这个时候就出现了shared\_ptr。

**shared\_ptr的实现原理**

shared\_ptr使用的引用计数的浅拷贝的形式，这个时候是不需要使用引用计数的写时拷贝的，因为多个指针指向的是同一个动态的内存空间，当其中的一个内存空间改变的时候，其他的内容也是相应的改变的。

这个时候我们的shared\_ptr这个类的成员变量中就要有一个指针，用于指向一个动态开辟的存储空间，还需要有一个用于计数的指针，这个指针指向一个动态开辟的内存空间，一般是整型的，这个整型 变量中存放的是我们指向同一个空间的个数，然后这个动态的整型空间只在构造函数的使用开辟出来，其他的拷贝构造函数还有赋值运算符的重载 的时候直接的使用了。

template<class T>

class Shared\_Ptr

{

public:

Shared\_Ptr(T\* ptr)

:\_ptr(ptr)

,\_count(new int(1))

{

}

Shared\_Ptr(const Shared\_Ptr& ptr)

:\_ptr(ptr.\_ptr)

, \_count(ptr.\_count)

{

(\*\_count)++;

}

Shared\_Ptr& operator=(const Shared\_Ptr& ptr)

{

if (this == &ptr)

{

return \*this;

}

else

{

if (!--(\*\_count))

{

delete \_ptr;

delete \_count;

cout << "delete older" << endl;

}

\_ptr = ptr.\_ptr;

\_count = ptr.\_count;

(\*\_count)++;

}

return \*this;

}

~Shared\_Ptr()

{

(\*\_count)--;

if ((\*\_count) == 0)

{

delete \_ptr;

delete \_count;

cout << "delete" << endl;

}

}

T& operator\*()

{

return \*\_ptr;

}

T\* operator->()

{

return \_ptr;

}

private:

T\* \_ptr;

int\* \_count;

};

代码分析

构造函数

当我们使用构造函数的时候，这个时候肯定是已经动态开辟了一个内存空间的，所以我们这个时候也给我们\_count指针动态的开辟一个空间，并且这个空间的值是1，因为此时一定是有一个空间了。

拷贝构造函数

拷贝构造函数的时候，我们是让一个已经存在的对象去初始化另一个对象，所以这个时候我们只需要让当前对象的指针指向那个动态的空间，同时时当前对象的计数指针也指向那个对象的计数空间，并且使当前对象的计数值加1，因为这个时候已经有两个对象 指向了一个动态的空间了。

析构函数

析构函数的时候，我们需要把引用计数减一，这个时候再去判断我们的引用计数的值是否为0，如果是0，这个时候就需要释放我们的动态管理的空间，同时释放掉我们的引用计数的动态空间，同时需要把这两个指针置为NULL。

赋值运算符的重载

该检查的是，这个赋值是不是自赋值,如果是 自赋值，这个时候直接返回该对象 即可了，还需要注意的是，我们 的引用计数不需要加1.如果不是自赋值，此时需要把当前对象的引用计数减1，同时判断减1之后应用计数是不是为0，如果是0则需要释放掉，如果不是0，就不要管了，然后接下来把我们当前指针指向被赋值的那个空间即可。

**循环引用和weak\_ptr**

当下面的代码的时候会出现一种情况就是 循环引用，首先看代码，下面的代码是我们定义的一个结构体

struct Str

{

Shared\_Ptr<Str> \_prev;

Shared\_Ptr<Str> \_next;

};

再来看我们的构造函数

Shared\_Ptr(T\* ptr)

:\_ptr(ptr)

,\_count(new int(1))

{

}

上面的构造函数中，我们的构造函数没有缺省值，会报错，因为我们在定义\_prev和\_next的时候，没有传入参数，所以我们需要把我们的构造函数改成下面的样子，就是把缺省值赋值为NULL。

这个时候我们写一个测试用例

Shared\_Ptr<Str> a = new Str;

Shared\_Ptr<Str> b = new Str;

这个时候会打印出六个delete，因为我们new出来的两个对象本身是有两个对象的，所以析构的时候，会析构这里面的两个一个是\_next一个是\_prev，然后我们的a和b本身也是指向对象的，所以一共析构了六次。

**隐患问题 -- 循环引用**

如果我们是像下面的方式 使用它，就会出现循环 引用的问题，请看下面的代码

Shared\_Ptr<Str> a = new Str;

Shared\_Ptr<Str> b = new Str;

a->\_next = b;

b->\_prev = a;

结合刚刚的分析，我们来分析上面的一段程序，首先是a和b分别指向了两个new出来的对象 ，然后这两个对象 里面的next和prev分别指向了两个对象，接着 执行a->\_next = b; b->\_prev = a;的时候，会调用赋值运算符的重载，然后就是a里面的next由原来的内容指向了b，b里面 的prev由原来的内容指向了a；这个时候问题就来了，b这个指向指向的内存空间有两个指针指向着，一个是b自己，一个是a->\_next，所以析构的时候不会释放内存空间，这不是我们想看到的结果。也可以这样子分析，就是我们的a和b析构的时候，只是 把引用计数减1，接下来析构a->\_next和b->prev的时候，都是相互依赖彼此的，所以都释放不了，这就是循环引用。

**weak\_ptr**

于是为了解决上面的循环引用的特殊场景，配合着shared\_ptr设计出了一个weak\_ptr，代码如下

#include<iostream>

using namespace std;

template<class T>

class Weak\_ptr;

template<class T>

class Shared\_Ptr

{

public:

friend class Weak\_ptr<T>;

Shared\_Ptr(T\* ptr = NULL)

:\_ptr(ptr)

,\_count(new int(1))

{

}

Shared\_Ptr(const Shared\_Ptr& ptr)

:\_ptr(ptr.\_ptr)

, \_count(ptr.\_count)

{

(\*\_count)++;

}

Shared\_Ptr& operator=(Shared\_Ptr& ptr)

{

if (this == &ptr)

{

return \*this;

}

else

{

if (!--(\*\_count))

{

delete \_ptr;

delete \_count;

cout << "delete older" << endl;

}

\_ptr = ptr.\_ptr;

\_count = ptr.\_count;

(\*\_count)++;

}

return \*this;

}

~Shared\_Ptr()

{

(\*\_count)--;

if ((\*\_count) == 0)

{

delete \_ptr;

delete \_count;

\_ptr = NULL;

\_count = NULL;

cout << "delete" << endl;

}

}

T& operator\*()

{

return \*\_ptr;

}

T\* operator->()

{

return \_ptr;

}

protected:

T\* \_ptr;

int\* \_count;

};

template<class T>

class Weak\_ptr

{

public:

Weak\_ptr()

:\_ptr(NULL)

{}

Weak\_ptr(Shared\_Ptr<T> ptr)

{

\_ptr = ptr.\_ptr;

}

T\* operator->()

{

return \_ptr;

}

private:

T\* \_ptr;

};

struct Str

{

Weak\_ptr<Str> \_prev;

Weak\_ptr<Str> \_next;

int \_a;

};

void TestPtr()

{

Shared\_Ptr<Str> a = new Str;

Shared\_Ptr<Str> b = new Str;

a->\_next = b;

b->\_prev = a;

a->\_next->\_a = 0;

cout << b->\_a;

}

首先我们看Weak\_ptr，它的成员变量是T\* \_ptr;此时我们的Str那个自定义的结构体中的指针就可以改成了Weak\_ptr的形式，因为我们的Weak\_ptr维护的是一个普通的指针，但是我们在使用的时候需要用到\_next指向一个Shared\_ptr的对象，所以这个时候，我们不要把Weak\_ptr的拷贝构造函数的参数写成是Shared\_ptr，然后赋值的时候，需要把Shared\_ptr的\_ptr赋值给Weak\_ptr的\_ptr，但是Shared\_ptr中的\_ptr是 私有的，所以这个时候我们在Shared\_ptr里面把Weak\_ptr声明为友元。但是这个时候又出现了一个 问题就是，我们的他声明为友元之后，编译器找不到我们的友元类，因为我们的Weak\_ptr的定义部分是在Shared\_ptr的后面，所以这个时候，需要在Shared\_ptr的前面声明我们的Weak\_ptr。

**定制防函数**

所谓的防函数就是让我们的类看起来像是函数一样

举一个简单的例子，看下面的代码

struct Compare

{

bool operator()(int a,int b)

{

return a > b;

}

};

void test()

{

Compare com;

cout << com(1,2);

}

类Compare是我们定制的一个防函数，下面的test就是我们把他 当成一个函数来使用它

为什么要引入防函数呢，因为我们在使用的时候上面的Shared\_ptr的时候，我们管理的内存 空间可能是一个FILE\*的一个指针，这个时候我们就不能只使用delete来释放我们的空间，这个时候我们就需要定制一个防函数，把它作为一个参数放在构造函数 中，同时我们的Shared\_ptr的成员变量里面需要定义一个这样的变量。

#include<iostream>

using namespace std;

#include<assert.h>

#pragma once

#define \_CRT\_SECURE\_NO\_WARNINGS 1

struct DelFile

{

void operator()(FILE\* f)

{

fclose(f);

cout << "fclose" << endl;

}

};

struct DelDel

{

void operator()(void\* p)

{

assert(p);

delete p;

cout << "delete" << endl;

}

};

template<class T,class Del>

class Weak\_ptr;

template<class T,class Del>

class Shared\_Ptr

{

public:

friend class Weak\_ptr<T,Del>;

Shared\_Ptr(T\* ptr,Del d)

:\_ptr(ptr)

,\_count(new int(1))

, \_del(d)

{

}

Shared\_Ptr(const Shared\_Ptr& ptr)

:\_ptr(ptr.\_ptr)

, \_count(ptr.\_count)

{

(\*\_count)++;

}

Shared\_Ptr& operator=(Shared\_Ptr& ptr)

{

if (this == &ptr)

{

return \*this;

}

else

{

if (!--(\*\_count))

{

del(\_ptr);

delete \_count;

cout << "delete older" << endl;

}

\_ptr = ptr.\_ptr;

\_count = ptr.\_count;

(\*\_count)++;

}

return \*this;

}

~Shared\_Ptr()

{

(\*\_count)--;

if ((\*\_count) == 0)

{

\_del(\_ptr);

delete \_count;

\_ptr = NULL;

\_count = NULL;

//cout << "delete" << endl;

}

}

T& operator\*()

{

return \*\_ptr;

}

T\* operator->()

{

return \_ptr;

}

protected:

T\* \_ptr;

int\* \_count;

Del \_del;

};

template<class T, class Del>

class Weak\_ptr

{

public:

Weak\_ptr()

:\_ptr(NULL)

{}

Weak\_ptr(Shared\_Ptr<T,Del> ptr)

{

\_ptr = ptr.\_ptr;

}

T\* operator->()

{

return \_ptr;

}

private:

T\* \_ptr;

};

struct Str

{

Weak\_ptr<Str,DelDel> \_prev;

Weak\_ptr<Str, DelDel> \_next;

int \_a;

};

void TestFile()

{

DelFile d;

Shared\_Ptr<FILE, DelFile> a(fopen("w.ss","w"),d); //注意这里传参的时候，首先要实例化一个对象

//我一开始使用的是Shared\_Ptr<FILE, DelFile> a(fopen("w.ss","w"),DelFile d)

//这种方式显然是错误的，我不能在一个函数里面去实例化一个对象

}

void TestPtr()

{

DelDel d;

Shared\_Ptr<Str, DelDel> a(new Str, d);

Shared\_Ptr<Str, DelDel> b(new Str, d);

a->\_next = b;

b->\_prev = a;

a->\_next->\_a = 0;

cout << b->\_a;

struct Compare

{

bool operator()(int a,int b)

{

return a > b;

}

};

void test()

{

Compare com;

cout << com(1,2);

}