# 02\_shared\_ptr

## 前言

智能指针shared\_ptr是存储动态创建对象的指针，其主要功能是管理动态创建对象的销毁，从而帮助彻底消除内存泄漏和悬空指针的问题。

## 实现原理

采用引用计数器的方法，允许多个智能指针指向同一个对象，当指向该对象的智能指针增加时，每增加一个，指向该对象的所有智能指针内部的引用计数加1，当指向该对象的智能指针减少时，每减少一个，引用计数会减1，当引用计数变为0，则表示没有shared\_ptr指针与此内存关联，此时使用delete函数删除该内存，自动释放动态分配的内存资源。

## 特点

1. 所指向的资源具有共享性，即多个shared\_ptr可以指向相同的内存。
2. shared\_ptr包含了一个指向对象的指针和一个指向控制块的指针。每一个由shared\_ptr管理的对象都有一个控制块，它除了包含强引用计数、弱引用计数之外，还包含了自定义删除器的副本和分配器的副本以及其他附加数据。
3. 智能指针将一个计数器与类指向的对象相关联，引用计数器跟踪共有多少个类对象共享同一指针
4. 每次创建类的新对象时，初始化指针并将引用计数置为1
5. 当对象作为另一对象的副本而创建时，拷贝构造函数拷贝指针并增加与之相应的引用计数
6. 对一个对象进行赋值时，赋值操作符减少左操作数所指对象的引用计数（如果引用计数为减至0，则删除对象），并增加右操作数所指对象的引用计数
7. 调用析构函数时，构造函数减少引用计数（如果引用计数减至0，则删除基础对象）

## 代码简单实现

template<typename T>

class SharedPtr

{

public:

SharedPtr(T\* ptr = NULL):\_ptr(ptr), \_pcount(new int(1))

{}

SharedPtr(const SharedPtr& s):\_ptr(s.\_ptr), \_pcount(s.\_pcount){

(\*\_pcount)++;

}

SharedPtr<T>& operator=(const SharedPtr& s){

if (this != &s)

{

if (--(\*(this->\_pcount)) == 0)

{

delete this->\_ptr;

delete this->\_pcount;

}

\_ptr = s.\_ptr;

\_pcount = s.\_pcount;

\*(\_pcount)++;

}

return \*this;

}

T& operator\*()

{

return \*(this->\_ptr);

}

T\* operator->()

{

return this->\_ptr;

}

~SharedPtr()

{

--(\*(this->\_pcount));

if (\*(this->\_pcount) == 0)

{

delete \_ptr;

\_ptr = NULL;

delete \_pcount;

\_pcount = NULL;

}

}

private:

T\* \_ptr;

int\* \_pcount;//指向引用计数的指针

};

## shared\_ptr的使用

### 构造函数创建

/\*\*\*1\*\*\*/

// ptr的意义就相当于一个指向nullptr的指针

shared\_ptr<T> ptr;

/\*\*\*2\*\*\*/

// 从new操作符的返回值构造

shared\_ptr<T> ptr(new T());

/\*\*\*3\*\*\*/

// 使用拷贝构造函数的方法，会让引用计数加1

// shared\_ptr可以当作函数的参数传递，或者当作函数的返回值返回，这个时候其实也相当于使用拷贝构造函数。

shared\_ptr<T> ptr2(ptr1);

/\*\*\*4\*\*\*/

// 假设B是A的子类

// 从shared\_ptr提供的类型转换(cast)函数的返回值构造

shared\_ptr<B> ptrb(new B());

shared\_ptr<A> ptra( dynamic\_pointer\_cast<A>(ptrb) );

/\*\*\*5\*\*\*/

// shared\_ptr的“赋值”

// a = b; 此后a原先所指的对象会被销毁，b所指的对象引用计数加1

// shared\_ptr也可以直接赋值，但是必须是赋给相同类型的shared\_ptr对象，

// 而不能是普通的C指针或new运算符的返回值。

// 当共享指针a被赋值成b的时候，如果a原来是NULL, 那么直接让a等于b，并且让它们指向的东西的引用计数加1;

// 如果a原来也指向某些东西的时候，如果a被赋值成b,

// 那么原来a指向的东西的引用计数被减1, 而新指向的对象的引用计数加1。

shared\_ptr<T> a(new T());

shared\_ptr<T> b(new T());

a = b;

/\*\*\*6\*\*\*/

// 已定义的共享指针指向新的new对象————reset()

// 原来所指的对象会被销毁

shared\_ptr<T> ptr(new T());

ptr.reset(new T());

### make\_shared辅助函数创建

std::shared\_ptr<int> foo = std::make\_shared<int> (10);

建议使用make\_shared的方式构造

### 自定义所指堆内存的释放规则

在初始化 shared\_ptr 智能指针时，还可以自定义所指堆内存的释放规则，这样当堆内存的引用计数为 0 时，会优先调用我们自定义的释放规则。

在某些场景中，自定义释放规则是很有必要的。比如，对于申请的动态数组来说，shared\_ptr 指针默认的释放规则是不支持释放数组的，只能自定义对应的释放规则，才能正确地释放申请的堆内存。

对于申请的动态数组，释放规则可以

1. 使用 C++11 标准中提供的 default\_delete 模板类

2. 可以自定义释放规则

//指定 default\_delete 作为释放规则

std::shared\_ptr<int> p6(new int[10], std::default\_delete<int[]>());

//自定义释放规则

void deleteInt(int\*p) {

delete []p;

}

//初始化智能指针，并自定义释放规则

std::shared\_ptr<int> p7(new int[10], deleteInt);

## shared\_ptr常用函数

get()函数，表示返回当前存储的指针（就是被shared\_ptr所管理的指针）。

但是不建议使用get()函数获取shared\_ptr关联的原始指针，因为如果在shared\_ptr析构之前手动调用了delete函数，会导致错误。

shared\_ptr<T> ptr(new T());

T \*p = ptr.get(); // 获得传统C指针

use\_count()函数，表示当前引用计数

shared\_ptr<T> a(new T());

a.use\_count(); //获取当前的引用计数

reset()函数，表示重置当前存储的指针。

shared\_ptr<T> a(new T());

a.reset(); // 此后 a 原先所指的对象会被销毁，并且 a 会变成 NULL

operator\*，表示返回对存储指针指向的对象的引用。它相当于：\* get（）。

operator->，表示返回指向存储指针所指向的对象的指针，以便访问其中一个成员。跟get函数一样的效果。

## 总结

以上就是今天要讲的内容，后续会有更多内容。

## 参考资料

版权声明：本文参考了其他资料和CSDN博主的文章，遵循CC 4.0 BY-SA版权协议，现附上原文出处链接及本声明。

1. [https://blog.csdn.net/weixin\_67596609/article/details/130722926](https://blog.csdn.net/weixin_67596609/article/details/130722926" \t "https://blog.csdn.net/weixin_67596609/article/details/_blank)