# 02\_shared\_ptr

## 前言

std::unique\_ptr对其持有的资源具有独占性，而std::shared\_ptr持有的资源可以在多个std::shared\_ptr之间共享，每多一个std::shared\_ptr对资源的引用，资源引用计数将增加1，每一个指向该资源的std::shared\_ptr对象析构时，资源引用计数减1，最后一个std::shared\_ptr对象析构时，发现资源计数为0，将释放其持有的资源。多个线程之间，递增和减少资源的引用计数是安全的。（注意：这不意味着多个线程同时操作std::shared\_ptr引用的对象是安全的）。std::shared\_ptr提供了一个use\_count()方法来获取当前持有资源的引用计数。除了上面描述的，std::shared\_ptr用法和std::unique\_ptr基本相同。

## 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。

## 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

### 初始化

1. 使用new关键字初始化

std::shared\_ptr<int> ptr(new int(123));

不能将原始指针直接赋给一个智能指针。下面这种方法是错误的：

std::shared\_ptr<int> sp=new int(1);

1. 赋值初始化

std::shared\_ptr<int> ptr1(new int(1));

std::shared\_ptr<int> ptr2=ptr1;

1. 调用拷贝构造函数初始化

std::shared\_ptr<int> ptr3(ptr1); // 使用拷贝构造函数的方法，会让引用计数加1

1. reset函数初始化

std::shared\_ptr<int> ptr; // 此时指向nullptr

ptr.reset(new int(1));

1. make\_shared初始化

建议优先使用make\_shared的方式初始化，

auto ptr=make\_shared<int>(100);

shared\_ptr<int> ptr=make\_shared<int>(100);

// 相当于

shared\_ptr<int> ptr(new int(100));

### 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

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

use\_count()函数，表示当前引用计数

返回与当前智能指针对象在同一指针上共享所有权的shared\_ptr对象的数量，如果这是一个空的shared\_ptr，则该函数返回0。如果要用来检查use\_count是否为1，可以改用成员函数unique会更快。

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

// 输出引用计数

cout << "ptr.use\_count() = " << ptr.use\_count() << endl;

### reset()函数

reset()函数，表示重置当前存储的指针。

void reset (ponit p)

没有参数时，先将管理的计数器引用计数减一并将管理的指针和计数器置清零。有参数p时，先做面前没有参数的操作，再管理p的所有权和设置计数器。

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\*

element\_type& operator\\*()

重载指针的 \* 运算符，返回管理的指针指向的地址的引用。

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

### operator->

element\_type\* operator->()

重载指针的 -> 运算符，返回管理的指针，可以访问其成员。

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

### unique()

bool unique()

返回当前shared\_ptr对象是否不和其他智能指针对象共享指针的所有权，如果这是一个空的shared\_ptr，则该函数返回false。

int\* p = new int(10);

// sp1与sp2都指向p

shared\_ptr<int> sp1(p);

cout << "sp1.unique() = " << sp1.unique() << endl; // 只有一个引用时是独占的返回true

shared\_ptr<int> sp2 = sp1;

cout << "sp1.unique() = " << sp1.unique() << endl; // 多个引用时不是独占的返回false

## shared\_ptr使用要注意的问题

### 问题1

不要用一个原始指针初始化多个shared\_ptr。

int \*ptr=new int;

shared\_ptr<int> p1(ptr);

shared\_ptr<int> p2(ptr);//逻辑错误，报错

### 问题2

不要在函数实参中创建shared\_ptr

// 错误示范

function(shared\_ptr<int>(new int),g());

因为C++的函数参数的计算顺序在不同的编译器不同的约定下可能是不一样的，一般是从右到左，但也可能从左到右；

所以，可能的过程是先new int，然后调用g()，如果恰好g()发生异常，而shared\_ptr还没有创建，则int内存泄漏；

正确的写法应该是先创建智能指针，代码如下：

shared\_ptr<int> p1(new int);

function(p1,g());

### 问题3

不要将this指针作为shared\_ptr返回回来，因为this指针本质上是一个裸指针，因此，可能会导致重复析构

#include <iostream>

#include <memory>

using namespace std;

class MyClass

{

public:

shared\_ptr<MyClass> GetSelf() {

return shared\_ptr<MyClass>(this);//不要这样做

}

MyClass() {

cout << "MyClass()" << endl;

};

~MyClass() {

cout << "~MyClass()" << endl;

};

};

int main()

{

// sp1与sp2都会调用new MyClass的析构函数，一个对象析构两次

shared\_ptr<MyClass> sp1(new MyClass);

shared\_ptr<MyClass> sp2 = sp1->GetSelf();

return 0;

}

在这个例子中，由于用同一个指针（this)构造了两个智能指针sp1和sp2，而他们之间是没有任何关系的，在离开作用域之后this将会被构造的两个智能指针各自析构，导致重复析构的错误。

正确返回this的shared\_ptr的做法是：让目标类继承std::enable\_shared\_from\_this类，然后使用基类的成员函数shared\_from\_this()返回this的shared\_ptr，如下所示：

#include <iostream>

#include <memory>

using namespace std;

// 继承std::enable\_shared\_from\_this<T>

class MyClass : public std::enable\_shared\_from\_this<MyClass>

{

public:

shared\_ptr<MyClass> GetSelf()

{

return shared\_from\_this();

}

MyClass()

{

cout << "MyClass()" << endl;

};

~MyClass()

{

cout << "~MyClass()" << endl;

};

};

int main()

{

shared\_ptr<MyClass> sp1(new MyClass);

shared\_ptr<MyClass> sp2 = sp1->GetSelf();

return 0;

}

### 问题4

避免循环引用，会导致内存泄漏。例如：

// 循环引用导致ap和bp的引用计数为2，在离开作用域之后，ap和bp的引用计数减为1，并不会减为0，导致两个指针都不会被析构，产生内存泄漏。

// 解决的办法是把A和B任何一个成员变量改为weak\_ptr。

#include <iostream>

#include <memory>

using namespace std;

class A;

class B;

class B

{

public:

shared\_ptr<A> aptr;

B();

~B();

private:

};

B::B()

{

cout << "B()" << endl;

}

B::~B()

{

cout << "B is deleted" << endl;

}

class A

{

public:

shared\_ptr<B> bptr;

A();

~A();

private:

};

A::A()

{

cout << "A()" << endl;

}

A::~A()

{

cout << "A is deleted" << endl;

}

int main()

{

shared\_ptr<A> ap(new A);

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

ap->bptr = bp;

bp->aptr = ap;

cout << "main leave" << endl; // 循环引用导致ap bp退出了作用域都没有析构

return 0;

}

## 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/haiguangboy/article/details/122278954
2. https://blog.csdn.net/LCZ411524/article/details/143648637
3. [https://blog.csdn.net/LCZ411524/article/details/143648637](https://blog.csdn.net/LCZ411524/article/details/143648637" \t "https://blog.csdn.net/LCZ411524/article/details/_blank)