# 走入smart\_pointer

## 前言

在使用C++编程时经常出现这种情况，申请的空间在函数结束忘记释放，造成内存泄漏。使用智能指针可以很大程度上的避免这个问题。

智能指针是用于存储动态分配对象指针的类，能自动释放对象，避免堆内存泄漏。它将动态分配资源交由类对象管理，类对象生命周期结束时，析构函数自动释放资源。所以智能指针管理内存的原理就是在函数结束时自动释放内存空间不需要手动释放内存空间。

C+ + 里面的四个智能指针：

1. auto\_ptr（C++11已弃用）
2. shared\_ptr,
3. weak\_ptr,
4. unique\_ptr

## unique\_ptr

unique\_ptr采用的是独享所有权语义，一个非空的unique\_ptr总是拥有它所指向的资源。转移一个 unique\_ptr将会把所有权全部从源指针转移给目标指针，源指针被置空；所以unique\_ptr不支持普通的拷贝和赋值操作，不能用在STL标准容器中；局部变量的返回值除外（因为编译器知道要返回的对象将要 被销毁）；如果你拷贝一个unique\_ptr，那么拷贝结束后，这两个unique\_ptr都会指向相同的资源，造成 在结束时对同一内存指针多次释放而导致程序崩溃。

## weak\_ptr

weak\_ptr：弱引用。 引用计数有一个问题就是互相引用形成环（环形引用），这样两个指针指向的内存 都无法释放。需要使用weak\_ptr打破环形引用。weak\_ptr是一个弱引用，它是为了配合shared\_ptr而引入 的一种智能指针，它指向一个由shared\_ptr管理的对象而不影响所指对象的生命周期，也就是说，它只引 用，不计数。如果一块内存被shared\_ptr和weak\_ptr同时引用，当所有shared\_ptr析构了之后，不管还有没 有weak\_ptr引用该内存，内存也会被释放。所以weak\_ptr不保证它指向的内存一定是有效的，在使用之 前使用函数lock()检查weak\_ptr是否为空指针。

## auto\_ptr

主要是为了解决“有异常抛出时发生内存泄漏”的问题 。因为发生异常而无法正常释放内存。 auto\_ptr有拷贝语义，拷贝后源对象变得无效，这可能引发很严重的问题；而unique\_ptr则无拷贝语义， 但提供了移动语义，这样的错误不再可能发生，因为很明显必须使用std::move()进行转移。 auto\_ptr不支持拷贝和赋值操作，不能用在STL标准容器中。STL容器中的元素经常要支持拷贝、赋值操 作，在这过程中auto\_ptr会传递所有权，所以不能在STL中使用。

说说你了解的auto\_ptr作用

1) auto\_ptr的出现，主要是为了解决“有异常抛出时发生内存泄漏”的问题；抛出异常，将导致指针p所指

向的空间得不到释放而导致内存泄漏；

2) auto\_ptr构造时取得某个对象的控制权，在析构时释放该对象。我们实际上是创建一个auto\_ptr类型的

局部对象，该局部对象析构时，会将自身所拥有的指针空间释放，所以不会有内存泄漏；

3) auto\_ptr的构造函数是explicit，阻止了一般指针隐式转换为 auto\_ptr的构造，所以不能直接将一般类型

的指针赋值给auto\_ptr类型的对象，必须用auto\_ptr的构造函数创建对象；

4) 由于auto\_ptr对象析构时会删除它所拥有的指针，所以使用时避免多个auto\_ptr对象管理同一个指针；

5) Auto\_ptr内部实现，析构函数中删除对象用的是delete而不是delete[]，所以auto\_ptr不能管理数组；

[capture] （parameters） mutable ->return-type {statement};118

6) auto\_ptr支持所拥有的指针类型之间的隐式类型转换。

7) 可以通过\*和->运算符对auto\_ptr所有用的指针进行提领操作；

8) T\* get(),获得auto\_ptr所拥有的指针；T\* release()，释放auto\_ptr的所有权，并将所有用的指针返回。

智能指针的循环引用

循环引用是指使用多个智能指针share\_ptr时，出现了指针之间相互指向，从而形成环的情况，有点类似

于死锁的情况，这种情况下，智能指针往往不能正常调用对象的析构函数，从而造成内存泄漏。举个例

子：

```c++

#include <iostream>

using namespace std;

template <typename T>

class Node

{

public:

Node(const T& value)

:\_pPre(NULL)

, \_pNext(NULL)

, \_value(value)

{

cout << "Node()" << endl;

}

~Node()

{

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

cout << "this:" << this << endl;

}

shared\_ptr<Node<T>> \_pPre;

shared\_ptr<Node<T>> \_pNext;

T \_value;

};

void Funtest()

{

shared\_ptr<Node<int>> sp1(new Node<int>(1));

shared\_ptr<Node<int>> sp2(new Node<int>(2));

cout << "sp1.use\_count:" << sp1.use\_count() << endl;

cout << "sp2.use\_count:" << sp2.use\_count() << endl;

sp1->\_pNext = sp2; //sp1的引用+1

sp2->\_pPre = sp1; //sp2的引用+1

cout << "sp1.use\_count:" << sp1.use\_count() << endl;

cout << "sp2.use\_count:" << sp2.use\_count() << endl;

}

int main()

{

Funtest();

system("pause");

return 0;

}

//输出结果

//Node()

//Node()

//sp1.use\_count:1119

//sp2.use\_count:1

//sp1.use\_count:2

//sp2.use\_count:2120

```

从上面shared\_ptr的实现中我们知道了只有当引用计数减减之后等于0，析构时才会释放对象，而上述情

况造成了一个僵局，那就是析构对象时先析构sp2,可是由于sp2的空间sp1还在使用中，所以sp2.use\_count

减减之后为1，不释放，sp1也是相同的道理，由于sp1的空间sp2还在使用中，所以sp1.use\_count减减之

后为1，也不释放。sp1等着sp2先释放，sp2等着sp1先释放,二者互不相让，导致最终都没能释放，内存泄

漏。

在实际编程过程中，应该尽量避免出现智能指针之前相互指向的情况，如果不可避免，可以使用使用弱

指针——weak\_ptr，它不增加引用计数，只要出了作用域就会自动析构。

手写实现智能指针类需要实现哪些函数？

1) 智能指针是一个数据类型，一般用模板实现，模拟指针行为的同时还提供自动垃圾回收机制。它会

自动记录SmartPointer<T\*>对象的引用计数，一旦T类型对象的引用计数为0，就释放该对象。

除了指针对象外，我们还需要一个引用计数的指针设定对象的值，并将引用计数计为1，需要一个构造

函数。新增对象还需要一个构造函数，析构函数负责引用计数减少和释放内存。

通过覆写赋值运算符，才能将一个旧的智能指针赋值给另一个指针，同时旧的引用计数减1，新的引用

计数加1

2) 一个构造函数、拷贝构造函数、复制构造函数、析构函数、移动函数；

智能指针出现循环引用怎么解决？

弱指针用于专门解决shared\_ptr循环引用的问题，weak\_ptr不会修改引用计数，即其存在与否并不影响对

象的引用计数器。循环引用就是：两个对象互相使用一个shared\_ptr成员变量指向对方。弱引用并不对对

象的内存进行管理，在功能上类似于普通指针，然而一个比较大的区别是，弱引用能检测到所管理的对

象是否已经被释放，从而避免访问非法内存。

//sp2.use\_count:1

//sp1.use\_count:2

//sp2.use\_count:2120

智能指针的作用

1) C++11中引入了智能指针的概念，方便管理堆内存。使用普通指针，容易造成堆内存泄露（忘记释

放），二次释放，程序发生异常时内存泄露等问题等，使用智能指针能更好的管理堆内存。

2) 智能指针在C++11版本之后提供，包含在头文件<memory>中，shared\_ptr、unique\_ptr、weak\_ptr。

shared\_ptr多个指针指向相同的对象。shared\_ptr使用引用计数，每一个shared\_ptr的拷贝都指向相同的内

存。每使用他一次，内部的引用计数加1，每析构一次，内部的引用计数减1，减为0时，自动删除所指

向的堆内存。shared\_ptr内部的引用计数是线程安全的，但是对象的读取需要加锁。

3) 初始化。智能指针是个模板类，可以指定类型，传入指针通过构造函数初始化。也可以使用

make\_shared函数初始化。不能将指针直接赋值给一个智能指针，一个是类，一个是指针。例如

std::shared\_ptr<int> p4 = new int(1);的写法是错误的

拷贝和赋值。拷贝使得对象的引用计数增加1，赋值使得原对象引用计数减1，当计数为0时，自动释放

内存。后来指向的对象引用计数加1，指向后来的对象

4) unique\_ptr“唯一”拥有其所指对象，同一时刻只能有一个unique\_ptr指向给定对象（通过禁止拷贝语

义、只有移动语义来实现）。相比与原始指针unique\_ptr用于其RAII的特性，使得在出现异常的情况

下，动态资源能得到释放。unique\_ptr指针本身的生命周期：从unique\_ptr指针创建时开始，直到离开作

用域。离开作用域时，若其指向对象，则将其所指对象销毁(默认使用delete操作符，用户可指定其他操

作)。unique\_ptr指针与其所指对象的关系：在智能指针生命周期内，可以改变智能指针所指对象，如创

建智能指针时通过构造函数指定、通过reset方法重新指定、通过release方法释放所有权、通过移动语义

转移所有权。

5) 智能指针类将一个计数器与类指向的对象相关联，引用计数跟踪该类有多少个对象共享同一指针。

每次创建类的新对象时，初始化指针并将引用计数置为1；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；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；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）。

6) weak\_ptr 是一种不控制对象生命周期的智能指针, 它指向一个 shared\_ptr 管理的对象. 进行该对象的内

存管理的是那个强引用的 shared\_ptr. weak\_ptr只是提供了对管理对象的一个访问手段。weak\_ptr 设计的

目的是为配合 shared\_ptr 而引入的一种智能指针来协助 shared\_ptr 工作, 它只可以从一个 shared\_ptr 或另

一个 weak\_ptr 对象构造, 它的构造和析构不会引起引用记数的增加或减少.

## 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)