问题

智能指针是C++11中的一个很重要的新特性，理解智能指针的用法无论对面试还是平时编写代码都有很大的好处，

所以接下来就让我们了解一下C++中的4种智能指针是如何工作的吧。

智能指针

C++的标准模板库（STL）中提供了4种智能指针：`auto_ptr`、`unique_ptr`、`share_ptr`、`weak_ptr`，其中后面3种是C++11的新特性，而 `auto_ptr` 是C++98中提出的，已经被C++11弃用了，取而代之的是更加安全的 `unique_ptr`，后面会详细介绍。

为什么要使用智能指针？

使用智能指针主要的目的是**为了更安全且更加容易地管理动态内存**。因为即使是一名经验丰富的程序员，也难免有时候会忘记释放之前申请的动态内存，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，因为智能指针就是一个类，当超出了类的作用域时，类会自动调用析构函数来会自动释放资源，不需要手动地释放内存。当然，这并不是说使用智能指针就不会发生内存泄漏，只是它在很大程度上可以防止由于程序员的疏忽造成的内存泄漏问题。

1. auto_ptr

智能指针 `auto_ptr` 由C++98引入，定义在头文件 `<memory>` 中，在C++11中已经被弃用了，因为它不够安全，而且可以被 `unique_ptr` 代替。那它为什么会被 `unique_ptr` 代替呢？先看下面这段代码：

```
1  #include <iostream>
2  #include <string>
3  #include <memory>
4  using namespace std;
5
6  int main() {
7      auto_ptr<string> p1(new string("hello world."));
8      auto_ptr<string> p2;
9      p2 = p1;                 //p2接管p1的所有权
10     cout << *p2<< endl;      //正确，输出：hello world.
11     //cout << *p1 << endl;   //程序运行到这里会报错
12
13     //system("pause");
14     return 0;
15 }
16
```

由上面可以看到，当第9行代码执行赋值操作后，`p2` 接管了 `p1` 的所有权，但此时程序并不会报错，而且运行第10行代码时可以正确输出指向对象的内容。而第11行注释掉的代码则会报错，因为 `p1` 此时已经是空指针了，访问输出它会使程序崩溃。可见，`auto_ptr` 智能指针并不够安全，于是有了它的替代方案：即 `unique_ptr` 指针。

2. unique_ptr

`unique_ptr` 同 `auto_ptr` 一样也是采用所有权模式，即同一时间只能有一个智能指针可以指向某个对象，但之所以说**使用 `unique_ptr` 智能指针更加安全**，是因为它相比于 `auto_ptr` 而言**禁止了拷贝操作**（`auto_ptr` 支持拷贝赋值，前面代码示例就是个例子），`unique_ptr` 采用了移动赋值 `std::move()` 函数来进行控制权的转移。例如：

```
#include <iostream>
#include <string>
#include <memory>
using namespace std;

int main() {
    unique_ptr<string> p1(new string("hello world"));
    //unique_ptr<string> p2(p1);     //编译不通过，禁止拷贝操作
    //unique_ptr<string> p2 = p1;    //编译不通过
    unique_ptr<string> p2 = std::move(p1);
    cout << *p2 << endl;
    //cout << *p1 <<endl;

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

```

由上面代码示例可知，`unique_ptr` 不会等到程序运行到访问 `p1` 的时候才崩溃掉，而是会在编译时就不给予通过，所以它相对而言更加安全。

当然，不管是 `auto_ptr` 还是 `unique_ptr`，都可以调用函数 `release` 或 `reset` 将一个（非 const）`unique_ptr` 的控制权转移给另一个 `unique_ptr`，如：

```
unique_ptr<string> p1(new string("hello world"));
unique_ptr<string> p2(p1.release());     //p2获得控制权，p1被置为空
unique_ptr<string> p3;
p3.reset(p2.release()); //p3获得p2的控制权，reset释放了p2原来的内存
```

此外，**`unique_ptr` 使用也很灵活，如果 `unique_ptr` 是个临时右值，编译器允许拷贝操作**。

```
unique_ptr<string> p2;
p2 = unique_ptr<string>(new string("hello world")); //正确，临时右值可以进行拷贝赋值操作
```

上面这种方式进行拷贝操作不会留下悬挂指针，因为它调用了 `unique_ptr` 的构造函数，该构造函数创建的临时对象在其所有权让给 `p2` 后就会被销毁。

3.share_ptr

共享指针 `share_ptr` 是一种可以共享所有权的智能指针，定义在头文件 `memory` 中，它允许多个智能指针指向同一个对象，并使用引用计数的方式来管理指向对象的指针（成员函数use_count()可以获得引用计数），该对象和其相关资源会在“最后一个引用被销毁”时候释放。

`share_ptr`是为了解决 `auto_ptr` 在对象所有权上的局限性(`auto_ptr` 是独占的)，在使用引用计数的机制上提供了可以共享所有权的智能指针。

> 引用计数变化的几种情况：
>
> ①在创建智能指针类的新对象时，初始化指针，引用计数设置为1；
>
> ②当智能指针类的对象作为另一个对象的副本时（即进行了拷贝操作），引用计数加1；
>
> ③当使用赋值操作符对一个智能指针类对象进行赋值时（如`p2=p1`），左操作数（即`p2`）引用先减1，因为它已经指向了别的地方，如果减1后引用计数为0，则释放指针所指对象的内存；然后右操作数（即`p1`）引用加1，因为此时左操作数指向了右操作数的对象。
>
> ④调用析构函数时，析构函数先使引用计数减1，若减至0则delete对象。

`share_ptr` 与内存泄漏：

当两个对象分别使用一个共享指针`share_ptr` 指向对方时，会导致内存泄漏的问题。

可以这样来理解：智能指针是用来管理指针的，当它指向某个对象时，该对象上的引用计数会加1，当引用计数减到0时该对象会销毁，也就是说**智能指针使用引用计数机制来管理着它所指对象的生命周期**，因此若某个对象A的`share_ptr` 指向了对象B，那么对象A只能在对象B先销毁之后它才会销毁；同理，若对象B的`share_ptr` 也指向了对象A，则只有在对象A先销毁之后B才会被销毁。因此，当两个对象的`share_ptr` 相互指向对方时，两者的引用计数永远不会减至0，即两个对象都不会被销毁，就会造成内存泄漏的问题。

4.weak_ptr

`weak_ptr` 弱指针是一种不控制对象生命周期的智能指针，它指向一个 `share_ptr` 管理的对象，进行该对象的内存管理的是那个强引用的 `share_ptr` ，也就是说 `weak_ptr` 不会修改引用计数，只是提供了一种访问其管理对象的手段，这也是它称为弱指针的原因所在。

此外，`weak_ptr` 和 `share_ptr` 之间可以相互转化，`share_ptr` 可以直接赋值给`weak_ptr` ，而`weak_ptr` 可以通过调用 `lock` 成员函数来获得`share_ptr` 。

```
//shared_ptr和weak_ptr代码示例
#include <iostream>
#include <memory>
using namespace std;

class B;
class A {
public:
    shared_ptr<B> _pb;
    //weak_ptr<B> _pb;
    ~A() { cout << "A delete" << endl; }
};
class B {
public:
    shared_ptr<A> _pa;
    ~B() { cout << "B delete" << endl; }
};

int main() {
    shared_ptr<B> pb(new B());  //创建一个B类对象的share_ptr
```

```
    shared_ptr<A> pa(new A());  //创建一个A类对象的share_ptr
    pb->_pa = pa;    //B指向A
    pa->_pb = pb;    //A指向B
    cout << pb.use_count() << endl; //打印结果为：2
    cout << pa.use_count() << endl; //打印结果为：2，可见两个指针相互引用了

    //system("pause");
    return 0;
}
//注：若将第9行代码注释掉，第10行取消注释，打印的结果将分别为：1 和 2
//当B析构函数时，其计数会变为0，B被释放，B释放的同时会使得A的计数减1，
//A再析构后A的计数也会变为0，得以释放
```

参考资料

牛客网-C++工程师面试宝典

C++ STL 四种智能指针