智能指针

为什么需要智能指针

所谓资源就是，一旦用了它，将来必须还给系统。C++中内存资源的动态分配经由new与delete实现。问题在于，无论是有意无意，我们有时候总会忘记释放内存中的资源。例如delete语句出现在某个循环语句中，而我们的continue或者break却跳过了它的执行；或者是在程序中某个分支含有函数return语句，而delete操作放在return 语句之后；更加难以预料的事情是程序执行过程中发生了异常，导致我们的delete语句没有执行。总的来说，**把资源回收交给用户并不是一种好做法。**我们期望有一种机制，它帮助我们管理从系统获取而来的资源，当我们不再使用该资源时，该机制能自动帮我们回收，避免了内存泄漏问题。智能指针就是这样一种资源回收机制。

智能指针具体是什么

《Effective C++》条款13提到，以对象来管理资源。这个条款提到了两个观点：

1. 获得资源后立刻放进管理对象内。
2. 管理对象运行析构函数确保资源被释放。

智能指针就是这样的一种类。它们的行为类似于指针，同样支持解引用* 或取成员->运算。智能指针将基本内置类型指针封装为类对象指针，管理着该指针所指向的动态分配资源，并通过类的析构函数对资源进行释放。在C++中，智能指针都是模板类，因为它们要管理的可能是用户自定义类型所分配的内存空间。

智能指针的实现原理

在STL中，一共是有四种智能指针：auto_ptr,unique_ptr,shared_ptr,weak_ptr。其中auto_ptr是C++98提供的智能指针，现在基本已经被弃用。原因后面有说。
其中auto_ptr,unique_ptr是独占型的智能指针。这里以auto_ptr为例，在某个时刻下，只能有一个auto_ptr指向一个给定的对象。shared_ptr则允许多个指针指向同一个对象，而weak_ptr指向的是shared_ptr所管理的对象，它是一种弱引用。
shared_ptr的实现基于引用计数技术。智能指针管理的着一个对象，并记录着所有管理同个对象的指针个数，这个个数称为计数。藉由智能指针去初始化或赋值其他智能指针时，计数的值增加1，表示资源对象多了一个引用；当智能指针的生命周期结束或者指向别的对象时，计数的值减去1，表示资源对象减少一个引用。智能指针生命周期结束时，编译器会调用它的析构函数，在析构函数中判断引用计数的值是否为0，若为0，则释放所管理的对象资源；若不为0，表明还有其他指针指向所管理的对象，不释放该对象资源。

auto_ptr / unique_ptr

上面说到auto_ptr是C++98提供的智能指针，现在已经被摒弃，原因在于**为了维护独占性，auto_ptr进行了不正常的复制/赋值行为。**
我们的赋值操作在语义上保证了右操作数不会在赋值时受到修改，然而，为了保证auto_ptr的独占性，这种语义被修改了。

```
auto_ptr<int> p1(new int(1));  /*1*/
auto_ptr<int> p2(p1); /*2*/
auto_ptr<int> p3;
p3= p2;/*3*/
```

p1 开始管理着值为1的对象，执行了2之后，p1被置空，由p2独占对象资源；执行3之后，p2被置为空，由p3独占对象资源。想象有一个元素为auto_ptr的数组：

```
    auto_ptr<int>vec[5]=
    {
        auto_ptr<int>(new int(1)),
        auto_ptr<int>(new int(2)),
        auto_ptr<int>(new int(3)),
        auto_ptr<int>(new int(4)),
        auto_ptr<int>(new int(5)),
    };
    for (auto & t : vec)
    {
        cout << *t << endl;
    }
    //vec[2]被置空
    auto_ptr<int> aptr = vec[2];
   //程序运行奔溃
    for (auto & t : vec)
    {
        cout << *t << endl;
    }
```

而我们的STL容器要求其元素可以有正常的复制行为，因此，**STL容器容不得auto_ptr**。而C++11新出现的智能指针unique_ptr比auto_ptr更聪明好用，unique_ptr拒绝直接的复制/赋值操作，必须通过reset/release接口来进行对象管理权的转移，这无疑提高了安全性；unique_ptr的聪明还体现在：

```
unique_ptr test()
{
    unique_ptr <int> temp(new int (1));
    return temp;
}

unique_ptr<int> p;
p = test();
```

在这里test返回的临时变量对p的赋值操作成功，因为临时变量复制结束后就被销毁，没有机会通过临时的unique_ptr对象去访问无效数据，这种赋值是安全的。
总结一下：

1. auto_ptr不适用于STL容器，且易造成对无效指针的访问导致程序奔溃。
2. unique_ptr比auto_ptr更加智能，安全性更高，应该选择使用unique_ptr。

weak_ptr有何作用

weak_ptr是一种不控制所指向对象生命期的智能指针，它指向由一个shared_ptr管理的对象。讲一个weak_ptr绑定到一个shared_ptr不会改变shared_ptr的引用计数，一旦最后一个指向对象的shared_ptr被销毁，对象就会被释放，即使有weak_ptr指向对象，对象还是会被释放。weak_ptr也取名自这种弱共享对象的特点。

相对于weak_ptr来说 ，shared_ptr是一种强引用的关系。在循环引用的情况下资源得不到回收，将造成内存泄漏。如下图出现了引用计数的循环引用问题：对象A被对象B所引用，对象C被对象A所引用，对象B被对象C所引用，这时每个对象的引用计数都是1，都在等待在引用它的对象释放对象，造成一种循环等待的现象，而资源也不会被如愿释放掉。

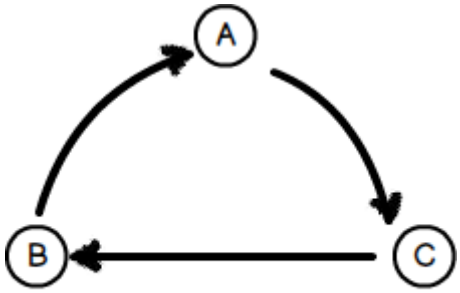

图. 循环引用

weak_ptr弱引用的出现正是能够打破这种循环引用。由于弱引用不更改引用计数，类似普通指针，只要把循环引用的一方使用弱引用，即可解除循环引用。虽然通过弱引用指针可以有效的解除循环引用，但这种方式必须在程序员能预见会出现循环引用的情况下才能使用，也可以是说这个仅仅是一种编译期的解决方案，如果程序在运行过程中出现了循环引用，还是会造成内存泄漏的。因此，即使使用了智能指针，C++还是无法完全杜绝内存泄漏的问题。