C++标准库智能指针(std::auto_ptr)

智能指针两大特性：
1.构造栈对象的生命期控制堆上构造的对象的生命期
2.通过 release 来保证 auto_ptr 对对象的独权.
另必须使用显示构造

文章结构：

一、剖析 C++标准库智能指针(std::auto_ptr)

1.Do you Smart Pointer?

Smart Pointer,中文名：智能指针, 舶来品?
不可否认,资源泄露(resource leak)曾经是 C++程序的一大噩梦.垃圾回收机制(Garbage Collection)一时颇受注目.然而垃圾自动回收机制并不能满足内存管理的即时性和可视性,往往使高傲的程序设计者感到不自在.
况且,C++实现没有引入这种机制.在探索中,C++程序员创造了锋利的"Smart Pointer".一定程度上,解决了资源泄露问题.

也许,经常的,你会写这样的代码：

```
//x 拟为 class:
//        class x{
//        public:
//            int m_Idata;
//        public:
//            x(int m_PARAMin):m_Idata(m_PARAMin){}
//            void print(){cout<<m_Idata<<endl;}
//        .....
//        }
//
void fook(){
x* m_PTRx = new A(m_PARAMin);
m_PTRx->DoSomething();    //#2
delete m_PTRx;
}
```

是的,这里可能没什么问题.可在复杂、N 行、m_PTRclassobj 所指对象生命周
期要求较长的情况下,你能保证你不会忘记 delete m_PTRclassobj 吗?生活中,
我们往往不应该有太多的口头保证,我们需要做些真正有用的东西.还有一个
更敏感的问题:异常.假如在#2 方法执行期异常发生,函数执行终止,那么 new 出的对象就会泄露.于是,你可能会说:那么就捕获异常来保证安全性好了.
你写这样的程式:

```
void fook(){
A* m_PTRx = new A(m_PARAMin);
try{
  m_PTRx->DoSomething();
}
catch(..){
  delete m_PTRx;
  throw;
}
delete m_PTRx;
}
```

哦!天哪!想象一下,你的系统,是否会象专为捕获异常而设计的.

一天,有人给你建议:"用 Smart Pointer,那很安全.".你可以这样重写你的程序:

```
void fook(){
auto_ptr<x> m_SMPTRx(new x(m_PARAMin));
m_SMPTRx->DoSomething();
}
```

OK!你不太相信.不用 delete 吗?
是的.不用整天提心吊胆的问自己:"我全部 delete 了吗?",而且比你的 delete
策略更安全.

然后,还有人告诉你,可以这样用呢:

```
ok1.
auto_ptr<x> m_SMPTR1(new x(m_PARAMin));
auto_ptr<x> m_SMPTR2(m_SMPTR1);  //#2
May be you can code #2 like this :
  auto_ptr<x> m_SMPTR2;
  m_SMPTR2 = m_SMPTR1;
ok2.
auto_ptr<int> m_SMPTR1(new int(32));
```

ok3.
auto_ptr<int> m_SMPTR1;
m_SMPTR1 = auto_ptr<int>(new int(100));
也可以：
auto_ptr<int> m_SMPTR1(auto_ptr<int>(new int(100)));

ok4.
auto_ptr<x> m_SMPTR1(new x(m_PARAMin));
m_SMPTR1.reset(new x(m_PARAMin1));

ok5.
auto_ptr<x> m_SMPTR1(new x(m_PARAMin));
auto_ptr<x> m_SMPTR2(m_SMPTR.release());
cout<<(*m_SMPTR2).m_Idata<<endl;

ok6.
auto_ptr<int> fook(){
return auto<int>(new int(100));
}

ok7.....and so on

但不可这样用：

no1.
char* chrarray = new char[100];
strcpy(chrarray,"I am programming.");
auto_ptr<char*> m_SMPTRchrptr(chrarray);
//auto_ptr 并不可帮你管理数组资源

no2.
vector<auto_ptr<x>> m_VECsmptr;
m_VECsmptr.push_back(auto_ptr<int>(new int(100)));
//auto_ptr 并不适合 STL 内容.

no3.
const auto_ptr<x> m_SMPTR1(new x(100));//所有权问题，不能用 const 类型
auto_ptr<x> m_SMPTR(new x(200));

no4.
x m_OBJx(300);
auto_ptr<x> m_SMPTR(&m_OBJx);//不支持栈资源的释放,因为析构中用的是

delete

no5
x* m_PTR = new x(100);
auto_ptr<x> m_SMPTR = m_pTR;//禁止隐式转换

no6.....and so on

预先提及所有权的问题,以便下面带着疑问剖析代码?

power1.
auto_ptr<x> m_SMPTR1(new x(100));
auto_ptr<x> m_SMPTR2 = m_SMPTR1;
m_SMPTR2->print();
//输出：100.
m_SMPTR1->print();
//!! 非法的.

power2.
auto_ptr<x> m_SMPTR(new x(100));

auto_ptr<x> returnfun(auto_ptr<x> m_SMPTRin){
return m_SMPTRin;
}

auto_ptr<x> = returnfun(m_SMPTR); //#5

//在上面的#5 中,我要告诉你对象所有权转移了两次.
//什么叫对象所有权呢?

2. std::auto_ptr 的设计原理

上面的一片正确用法,它们在干些什么?
一片非法,它们犯了什么罪?
一片什么所有权转移,它的内部机智是什么?
哦!一头雾水?下面我们就来剖析其实现机制.
基础知识：
a.智能指针的关键技术：在于构造栈上对象的生命期控制堆上构造的对象的生命期.因为在智能指针的内部,存储着堆对象的指针,而且在构析函数中调用 delete 行为.
大致机构如下：
x* m_PTRx = new x(100);//#1
template<typename T>
auto_ptr{
private:

```
T* m_PTR;//维护指向堆对象的指针,在 auto_ptr 定位后
....    //它应该指向#1 构造的对象,即拥有所有权.
~auto(){delete m_PTR;}
....
}
```

b.所有权转移之说

上面曾有一非法的程式片段如下：

```
auto_ptr<x> m_SMPTR1(new x(100));
auto_ptr<x> m_SMPTR2 = m_SMPTR1;
m_SMPTR2->print();
//输出：100.
m_SMPTR1->print();
//!! 非法的.
```

按常理来说,m_SMPTR->print();怎么是非法的呢?

那是因为本来,m_SMPTR1 维护指向 new x(100)的指针,

可是 m_SMPTR2 = m_SMPTR1;auto_ptr 内部机制使得 m_SMPTR1 将对象的地址

传给 m_SMPTR2,而将自己的对象指针置为 0.

那么自然 m_SMPTR->print();失败.

这里程序设计者要负明显的职责的.

那么 auto_ptr 为什么采取这样的策略：保证所有权的单一性.

亦保证了系统安全性.

如果多个有全权的 auto_ptr 维护一个对象,那么在你消除一个 auto_ptr 时,将导致多个 auto_ptr 的潜在危险.

下面我们以SGI-STL的auto_ptr设计为样本(去掉了无关分析的宏),来剖析其原理.

```
#1  template <class _Tp> class auto_ptr {
#2  private:
#3  _Tp* _M_ptr;  //定义将维护堆对象的指针

#4  public:
#5  typedef _Tp element_type;  //相关类型定义
#6  explicit auto_ptr(_Tp* __p = 0) __STL_NOTHROW : _M_ptr(__p) {}
#7  auto_ptr(auto_ptr& __a) __STL_NOTHROW : _M_ptr(__a.release()) {}
#8  template <class _Tp1> auto_ptr(auto_ptr<_Tp1>& __a)
__STL_NOTHROW
                        : _M_ptr(__a.release()) {}
```

//#6、#7、#8 是 auto_ptr 构造函数的三个版本.

//#6 注释：传入对象的指针,构造 auto_ptr.explicit 关键字:禁止隐式转换.

//　　这就是 ok2 正确,而 no5(隐式转换)错误的原因.

//#7 注释：拷贝构造函数.

//　　传入 auto_ptr 实例,构造 auto_ptr. ok1、ok3 使用了这个构造式.

```
//      它是一个很关键的构造函数,在具体情况下,我们再分析
//#8 注释：auto_ptr 的模板成员,可在继承对象重载的基础上,实现特殊功能.
//
//  举例：
//  class A{ public:
//      virtual void fook(){cout<<"I am programming"<<endl;
//      /*.........*/                    };
//  class B : public A {
//      virtual void fook(){ cout<<"I am working"<<endl;
//      /*.........*/                    };
//  auto_ptr<A> m_SMPTRa(new A(33));//实质：
//  auto_ptr<B> m_SMPTRb(m_SMPTRa); //基类的指针可以赋给派生类的指针
//
//  auto_ptr<B> m_SMPTRb(new B(44));//实质：
//  auto_ptr<A> m_SMPTRa(m_SMPTRb); //派生类的指针不可赋给基类的指针
//
//  auto_ptr<A> m_SMPTRa(new B(33));  // ok!
//  m_SMPTRa->fook()将调用派生类 B 的 fook()
//  m_SMPTRa->A::fook()将调用基类 A 的 fook()
//
//  auto_ptr<B> m_SMPTRb(new A(33));  // wrong!
//
//
#9  auto_ptr& operator=(auto_ptr& __a) __STL_NOTHROW {
#10 if (&__a != this) { delete _M_ptr;  _M_ptr = __a.release(); }
#11 return *this;
#12 }

#13 template <class _Tp1>
#14 auto_ptr& operator=(auto_ptr<_Tp1>& __a) __STL_NOTHROW {
#15 if (__a.get() != this->get()) { delete _M_ptr; _M_ptr = __a.release(); }
#16 return *this;
#16 }
  //
  // #9~~#16 两个版本的指派函数.
  //      delete _M_ptr; 在指派前,销毁原维护的对象.
  //      _a.release() ; release 操作,详细代码参见#20~~#23.
  //              用于*this 获得被指派对象,
  //              且将原维护 auto_ptr 置空.
  //    no3 使用了第一种指派.
```

```
      //    而权限转移正是_a.release()的结果.

    #17 ~auto_ptr() __STL_NOTHROW { delete _M_ptr; }
      //构析函数.消除对象.注意这里对对象的要求!

    #17 _Tp& operator*() const __STL_NOTHROW {  return *_M_ptr; }
    #18 _Tp* operator->() const __STL_NOTHROW { return _M_ptr;  }
    #19 _Tp* get() const __STL_NOTHROW { return _M_ptr; }
      //
      //  操作符重载.
      //#17 注释：提领操作(dereference),获得对象.  见 ok5 用法.
      //#18 注释：成员运算符重载,返回对象指针.
      //#19 注释：普通成员函数.作用同于重载->运算符
      //
    #20 _Tp* release() __STL_NOTHROW {
    #21 _Tp* __tmp = _M_ptr;
    #22 _M_ptr = 0;
    #23 return __tmp;          }
      //上面已经详解

    #24 void reset(_Tp* __p = 0) __STL_NOTHROW {
    #25 delete _M_ptr;
    #26 _M_ptr = __p;                }
      //
      //传入对象指针,改变 auto_ptr 维护的对象
      //     且迫使 auto_ptr 消除原来维护的对象
      //     见 ok3 用法.

      // According to the C++ standard, these conversions are required.
Most
      // present-day compilers, however, do not enforce that
requirement---and,
      // in fact, most present-day compilers do not support the language
      // features that these conversions rely on.

      //下面这片段用于类型转化,目前没有任何编译器支持
      //具体技术细节不诉.

      #ifdef __SGI_STL_USE_AUTO_PTR_CONVERSIONS

    #27 private:
    #28 template<class _Tp1>
    #29 struct auto_ptr_ref { _Tp1* _M_ptr; auto_ptr_ref(_Tp1* __p) :
_M_ptr(__p) {}
                };
```

```
#30 public:
#31 auto_ptr(auto_ptr_ref<_Tp> __ref) __STL_NOTHROW
             : _M_ptr(__ref._M_ptr) {}
#32 template <class _Tp1>
#33 operator auto_ptr_ref<_Tp1>() __STL_NOTHROW
#34 { return auto_ptr_ref<_Tp>(this->release()); }
#35 template <class _Tp1> operator auto_ptr<_Tp1>() __STL_NOTHROW
#36 { return auto_ptr<_Tp1>(this->release()); }
#37 #endif /* __SGI_STL_USE_AUTO_PTR_CONVERSIONS */
#38 };
```

OK!就是这样了.
正如上面原理介绍处叙说,
你需要正视两大特性：
1.构造栈对象的生命期控制堆上构造的对象的生命期
2.通过 release 来保证 auto_ptr 对对象的独权.

在我们对源码分析的基础上,重点看看：
no 系列错误在何处?
no1.
我们看到构析函数 template<class _Tp>

```
~auto_ptr() _STL_NOTHROW
{ delete _M_ptr; }
```

所以它不能维护数组,
维护数组需要操作：delete[] _M_ptr;
no2.
先提部分 vector 和 auto_ptr 代码：
a.提 auto_ptr 代码

```
auto_ptr(auto_ptr& __a) __STL_NOTHROW : _M_ptr(__a.release()) {}
```

b.提 vector 代码

```
Part1:
void push_back(const _Tp& __x) {
if (_M_finish != _M_end_of_storage) {
construct(_M_finish, __x);
++_M_finish;
}
else
_M_insert_aux(end(), __x);
}

Part2:
```

```
template <class _T1, class _T2>
inline void construct(_T1* __p,

//+++++++++++++++++++++++++++++++
//      const _T2& __value) {+
//+++++++++++++++++++++++++++++++
//  new (__p) _T1(__value);    +
//+++++++++++++++++++++++++++++++

}
```

Part3.

```
template <class _Tp, class _Alloc>
void
vector<_Tp, _Alloc>::_M_insert_aux
(iterator __position,

 //+++++++++++++++++++++++++++++++
 //     const _Tp& __x)     ++
 //+++++++++++++++++++++++++++++++

{
if (_M_finish != _M_end_of_storage) {
construct(_M_finish, *(_M_finish - 1));
++_M_finish;

//+++++++++++++++++++++++++++++++
//    _Tp __x_copy = __x;     +
//+++++++++++++++++++++++++++++++

copy_backward(__position, _M_finish - 2, _M_finish - 1);
*__position = __x_copy;
}
else {
const size_type __old_size = size();
const size_type __len = __old_size != 0 ? 2 * __old_size : 1;
iterator __new_start = _M_allocate(__len);
iterator __new_finish = __new_start;
__STL_TRY {
__new_finish = uninitialized_copy
(_M_start, __position, __new_start);
construct(__new_finish, __x);
++__new_finish;
__new_finish = uninitialized_copy
(__position, _M_finish, __new_finish);
```

```
 }
 __STL_UNWIND((destroy(__new_start,__new_finish),
      _M_deallocate(__new_start,__len)));
destroy(begin(), end());
_M_deallocate(_M_start, _M_end_of_storage - _M_start);
_M_start = __new_start;
_M_finish = __new_finish;
_M_end_of_storage = __new_start + __len;
}
}
```

从提取的 vector 代码,Part1 可看出,push_back 的操作行为.
兵分两路,可是再向下看,你会发现,无一例外,都
通过 const _Tp& 进行拷贝行为,那么从 auto_ptr 提出的片段就
派上用场了.
可你知道的,auto_ptr 总是坚持对对象的独权.那必须修改
原来维护的对象,而 vector 行为要求 const _Tp&,这样自然会产生
问题.一般编译器是可以发觉这种错误的.

其实,STL 所有的容器类都采用 const _Tp&策略.

```
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+ 看了 sutter 和 Josuttis 的两篇文章中,都提及：                +
+STL 容器不支持 auto_ptr 原因在于 copy 的对象只是获得所有权的对象, +
+ 这种对象不符合 STL 的要求.可是本人总感觉即时不是真正的复制对象,+
+ 但我用 vector<auto_ptr<x> >的目的就在于维护对象,并不在乎     +
+ 所谓的完全对象.而且我用自己写的 Smart Pointer 配合 STL 容器工作, +
+ 很正常.那需要注意的仅仅是 const 问题.                      +
+                                  +
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

no3.
这个也是 auto_ptr 隐含的所有权问题引起的.
const auto_ptr 不允许修改.
随便提及：const 对象不代表对象一点不可以改变.
在两种 const 语义下,都有方法修改对象或对象内部指针维护的对象
或其它资源.
no4.
再看 auto_ptr 的构析函数.
delete 不可以消除栈上资源.

no5.
依赖传入对象指针的构造函数被声明为 explicit,禁止隐式转换.

3.auto_ptr 高级使用指南

a.类成员 auto_ptr,禁止构造函数以构建"完全对象"

Programme1:

```
struct Structx{
    int m_Idata;
    char m_CHRdata;
    /* and so on */
};
```

出于对象编程的理念,

我们将 Structx 打造成包裹类:

```
class StructWrapper{
private:
Structx* m_STRTxptr;
public:
StructWrapper():m_STRTxptr(new Structx){}
~StructWrapper(){delete m_SMRTxptr;}
public:
void Soperator1(){ /* 针对 Structx 对象的特性操作 */}
void Soperator2(){ /* 针对 Structx 对象的特性操作 */}
/*  and so on */
};
```

Programme2:

```
class StructWrapper{
private:
auto_ptr<Structx> m_SMPTRx;
public:
StructWrapper():m_SMPTRAx(new Structx){}
public:
void Soperator1(){ /* 针对 Structx 对象的特性操作 */}
void Soperator2(){ /* 针对 Structx 对象的特性操作 */}
/*  and so on */
};
```

Programme3:

```
StructWrapper::StructWrapper(const StructWrapper& other)
: M_SMPTRx(new Struct(*other.m_SMPTRx)) {}
StructWrapper& StructWrapper::operator=(const StructWrapper
&other){
*m_SMPTRx = *other.m_SMPTRx;
};
```

处于对构建于堆中的对象(new Structx)智能维护的需要.
我们将 programme1 改造为 programme2：
不错,对象是可以智能维护了.
对于包裹类(StructWrapper)你是否会有这样的构造或指派操作：
StructWrapper m_SMPTRWrapper2(m_SMPTRWrapper1);

StructWrapper mSMPTRWrapper2 = m_SMPTRWrapper1;
那么请注意：
当你坦然的来一个：M_SMPTRWrapper1->Soperator1();的时候,
系统崩溃了.
不必惊讶,所有权还是所有权问题.
问一下自己：当 programme2 默认拷贝构造函数作用时,又调用了 auto_ptr 的
默认构造函数,那么 auto_ptr 所有的默认行为都遵循独权策略.对,就这样.
m_SMPTRWrapper1 的对象所有权转移给了 m_SMPTRWrapper2.
M_SMPTRWrapper1->Soperator1();那么操作变成了在 NULL 上的.
哦!系统不崩溃才怪.
那么你需要想,programme3 那样利用 auto_ptr 的提领操作符自己的
构造"完全对象".

b.利用 const 关键字,防止不经意的权限转移

从上面的叙述,你可看出,所有权转移到处可以酿成大祸.
而对于一般应用来说,独权又是很好的安全性策略.
那么我们就用 const 来修饰 auto_ptr,禁止不经意的错误.

当然上面提及：并不代表 auto_ptr 是不可修改的.
处于需要,从两种 const 语义,你都可实现修改.

然,你还希望在函数传入传出 auto_ptr 那么你可传递 auto_ptr 的引用,
那就万无一失了：void fook(const auto_ptr<x>& m_PARAMin);
在返回后赋予其它时,使用引用是不行的.你得用指针.
因为引用无论作为 lvalue 还是 rvaluev,都会调用构造或指派函数.

4.你是否觉得 std::auto_ptr 还不够完美

在实践中,std::auto_ptr 能满足你的需求吗?

Andrei Alexandrescu 在一篇文章中,提及：有关 Smart Pointer 的技术就像
巫术.Smart Pointer 作为 C++垃圾回收机制的核心,它必须足够强大的、具有工业强度和安全性.
但为了可一劳永逸我们还需要披荆斩棘继续探索.

下面在需求层面上,我们思索一下我们的智能指针还需要些什么?

a. std::auto_ptr 能够处理数组吗?我们可以用智能指针来管理其它的资源吗?

譬如一个线程句柄、一个文件句柄 and so on！

b. 对于我们的对象真的永远实行独权政策吗?

c. Our 智能指针还需要在继承和虚拟层面上发挥威力 ！

d. 往往,需要扩展 Our 智能指针的功能成员函数来满足动态的需要 ！

e. 也许,你需要的还很多.

智能指针 std::auto_ptr 和 shared_ptr

auto_ptr 类可以用于管理由 new 分配的单个对象,但是无法管理动态分配的数组（我们通常不会使用数组，而是使用 vector 代替数组）。auto_ptr 在拷贝和赋值的时候有不寻常的行为，因此 auto_ptrs 不能被保存在 stl 的容器中。当 auto_ptr 离开了自己的作用域或者被销毁,由 auto_ptr 管理的对象也会被销毁。

使用 std::auto_ptr 需要的头文件: #include <memory>

```
// 示例 1(b): 安全代码, 使用了 auto_ptr
  void f()
  {
     auto_ptr<T> pt( new T );
  .....
  } // 酷: 当 pt 出了作用域时析构函数被调用，从而对象被自动删除
```

现在代码不会泄漏 T 类型的对象，不管这个函数是正常退出还是抛出了异常，因为 pt 的析构函数总是会在出栈时被调用。清理会自动进行。

最后，使用一个 auto_ptr 就像使用一个内建的指针一样容易，而且如果想要“撤销”资源，重新采用手动的所有权，我们只要调用 release().

```
// 示例 2: 使用一个 auto_ptr
void g()
{
T* pt1 = new T;  // 现在，我们有了一个分配好的对象
 auto_ptr<T> auto_pt2( pt1 ); // 将所有权传给了一个 auto_ptr 对象, auto_pt2 指向了 pt1

// 使用 auto_ptr 就像我们以前使用简单指针一样
auto_pt2 = 12; // 就像 "*pt1 = 12;"
auto_pt2->SomeFunc(); // 就像 "pt1->SomeFunc();"
```

```
    // 用 get()来获得指针的值
    assert( pt1 == auto_pt2.get() );  // 二者一样
    // 用 release()来撤销所有权， auto_pt2 把保存的指针地址给了 pt3,
而自己指向了 NUll。
    T* pt3 = auto_pt2.release();  //
    // 自己删除这个对象，因为现在没有任何 auto_ptr 拥有这个对象
    delete pt3;
    } // pt2 不再拥有任何指针，所以不要试图删除它...ok，不要重复删除
```

最后，我们可以使用 auto_ptr 的 reset()函数来重置 auto_ptr 使之拥有另一个对象。如果这个 auto_ptr 已 经拥有了一个对象，那么，它会先删除已经拥有的对象，因此调用 reset()就如同销毁这个 auto_ptr，然后新建一个并拥有一个新对象：

```
    // 示例 3: 使用 reset()
    //
    void h()
    {
    auto_ptr<T> pt( new T(1) );
    pt.reset( new T(2) );  //即 pt 会首先 delete pt 目前指向的地址（new T
(1)得到的地址),
                //然后再指向 new T(2)分配的地址
    } // 最后，pt 出了作用域，
    // 第二个 T 也被自动删除了
```

Boost 智能指针——shared_ptr

boost::scoped_ptr 虽然简单易用，但它不能共享所有权的特性却大大限制了其使用范围，而 boost::shared_ptr 可以解决这一局限。顾名思义，boost::shared_ptr 是可以共享所有权的智能指针，首先让我们通过一个例子看看它的基本用法：

```
#include <string>
#include <iostream>
#include <boost/shared_ptr.hpp>

class implementation
{
public:
  ~implementation() { std::cout <<"destroying implementation\n"; }
  void do_something() { std::cout << "did something\n"; }
};

void test()
{
  boost::shared_ptr<implementation> sp1(new implementation());
```

```
    std::cout<<"The Sample now has "<<sp1.use_count()<<" references\n";

    boost::shared_ptr<implementation> sp2 = sp1;
    std::cout<<"The Sample now has "<<sp2.use_count()<<" references\n";

    sp1.reset();
    std::cout<<"After Reset sp1. The Sample now has "<<sp2.use_count()<<" references\n";

    sp2.reset();
    std::cout<<"After Reset sp2.\n";
}

void main()
{
    test();
}
```

该程序的输出结果如下：

```
The Sample now has 1 references
The Sample now has 2 references
After Reset sp1. The Sample now has 1 references
destroying implementation
After Reset sp2.
```

可以看到，boost::shared_ptr 指针 sp1 和 sp2 同时拥有了 implementation 对象的访问权限，且当 sp1 和 sp2 都释放对该对象的所有权时，其所管理的的对象的内存才被自动释放。在共享对象的访问权限同时，也实现了其内存的自动管理。

boost::shared_ptr 的内存管理机制：

boost::shared_ptr 的管理机制其实并不复杂，就是对所管理的对象进行了引用计数，当新增一个 boost::shared_ptr 对该对象进行管理时，就将该对象的引用计数加一；减少一个 boost::shared_ptr 对该对象进行管理时，就 将该对象的引用计数减一，如果该对象的引用计数为 0 的时候，说明没有任何指针对其管理，才调用 delete 释放其所占的内存。

上面的那个例子可以的图示如下：

1. sp1 对 implementation 对象进行管理，其引用计数为 1
2. 增加 sp2 对 implementation 对象进行管理，其引用计数增加为 2
3. sp1 释放对 implementation 对象进行管理，其引用计数变为 1
4. sp2 释放对 implementation 对象进行管理，其引用计数变为 0，该对象被自动删除

boost::shared_ptr 的特点：

和前面介绍的 boost::scoped_ptr 相比，boost::shared_ptr 可以共享对象的所有权，因此其使用范围基本上没有什 么限制（还是有一些需要遵循的使用规则，下文中介绍），自然也可以使用在 stl 的容器中。另外它还是线程安全的，这点在多线程程序中也非常重要。

boost::shared_ptr 的使用规则：

boost::shared_ptr 并不是绝对安全，下面几条规则能使我们更加安全的使用 boost::shared_ptr：

1. 避免对 shared_ptr 所管理的对象的直接内存管理操作，以免造成该对象的重释放
2. shared_ptr 并不能对循环引用的对象内存自动管理（这点是其它各种引用计数管理内存方式的通病）。
3. 不要构造一个临时的 shared_ptr 作为函数的参数。

   如下列代码则可能导致内存泄漏：

```
void test()
{
   foo(boost::shared_ptr<implementation>(new    implementation()),g());
}
```

   正确的用法为：

```
void test()
{
   boost::shared_ptr<implementation> sp    (new implementation());
   foo(sp,g());
}
```