个人资料

zjufirefly

访问： 6422次
积分： 464
等级：
排名： 千里之外

原创： 42篇 转载： 4篇
译文： 0篇 评论： 1条

文章搜索

文章分类

C++ (14)
其他 (5)
链接 (13)
linux (13)
shell (7)
操作 (11)

文章存档

2014年12月 (8)
2014年11月 (7)
2014年10月 (8)
2014年09月 (6)
2014年08月 (4)

展开

阅读排行

git操作命令 (327)
blktrace (287)
mysql常用命令 (280)
python (270)
cgroups (262)
linux下如何配置openvpn (260)

为什么需要auto_ptr_ref

分类： C++ 2014-05-03 16:57 84人阅读 评论(1) 收藏 编辑 删除

这几天开始拜读侯捷先生[illegible]作《C++标准程序库：自修教程与参考手册》。两位先生确实译功上乘，读得很顺。但是读到P55页关于auto_ptr_ref的讨论，却百思不得其解：为什么需要引入auto_ptr_ref这个辅助类呢？

从书中描述来看，仿佛与拷贝构造函数 、右值 、类型转换 有关。于是，结合auto_ptr的源代码，google之、baidu之，找了一推资料，终于初步 搞清该问题。

auto_ptr的拥有权

C++常见的智能指针有std::auto_ptr、boost::shared_ptr、boost::scoped_ptr、boost::shared_array、boost::scoped_array等。auto_ptr只是其中一种而已。但是，为什么auto_ptr才有auto_ptr_ref ，而boost::shared_ptr却没有shared_ptr_ref呢？

答案与auto_ptr的特性有关。auto_ptr强调对资源的拥有权 （ownership）。也就是说，auto_ptr是"它所指对象"的拥有者。而一个对象只能属于一个拥有者，严禁一物二主，否则就是重婚罪，意料外的灾难将随之而来。

为了保证auto_ptr的拥有权唯一，auto_ptr的拷贝构造函数和赋值操作符做了这样一件事情：移除另一个**auto_ptr**的拥有权 。为了说明拥有权的转移 ，请看下面的代码示例：

```cpp
#include <iostream>
#include <memory>

using namespace std;

int main(int argc, char **argv){
    auto_ptr<int> ptr1(new int(1));
    auto_ptr<int> ptr2(ptr1);  //ptr1的拥有权被转移到ptr2

    auto_ptr<int> ptr3(NULL);
    ptr3 = ptr2;       //ptr2的拥有权被转移到ptr3

    cout<<ptr1.get()<<endl;   //结果为0
    cout<<ptr2.get()<<endl;   //结果为0
    cout<<*ptr3<<endl;        //结果为1<span style="font-
family: Verdana, Geneva, Arial, Helvetica, sans-serif;">}</span>
```

auto_ptr的拷贝构造函数与赋值操作符

由于需要实现拥有权的转移，auto_ptr的拷贝构造函数和赋值操作符，与一般类的做法不太相同。我们可以看看MinGW5.1.6实现的auto_ptr源代码：

```cpp
 /**
  *  @brief  An %auto_ptr can be constructed from another %auto_ptr.
  *  @param  a  Another %auto_ptr of the same type.
  *
  *  This object now @e owns the object previously owned by @a a,
  *  which has given up ownsership.
  */
auto_ptr(auto_ptr& __a) throw() : _M_ptr(__a.release()) {}

/**
  *  @brief  %auto_ptr assignment operator.
  *  @param  a  Another %auto_ptr of the same type.
  *
```

vim-ctags-taglist-netrw (249)
性能测试命令字段解释 (213)
正则表达式 (211)
SystemTap (195)

评论排行

为什么需要auto_ptr_ref (1)
valgrind内存检查 (0)
cgroups (0)
linux下如何配置openvpn (0)
git操作命令 (0)
thrift (0)
Apache开源软件 (0)
vim-ctags-taglist-netrw (0)
vim常用配置 (0)
shell按行读取字符串，并 (0)

推荐文章

* 挣扎与彷徨--我的2014
* 校招回忆录---小米篇
* Android UI-自定义日历控件
* 30岁程序员回顾人生、展望未来
* 2014年终总结，我决定要实现的三个目标
* Android 启动问题——黑屏 死机解决方法

最新评论

为什么需要auto_ptr_ref
zjufirefly: explicit auto_ptr_ref(_Tp1* __p): _M_ptr(__p) {}...

```
 *  This object now @e owns the object previously owned by @a a,
 *  which has given up ownsership.  The object that this one @e
 *  used to own and track has been deleted.
 */
auto_ptr&
operator=(auto_ptr& __a) throw () {
    reset(__a.release());
    return *this;
}
```

可以看到，auto_ptr的拷贝构造函数、赋值操作符，它们的参数都是auto_ptr& ，而不是auto_ptr const & 。

一般来说，类的拷贝构造函数和赋值操作符的参数都是const &。但是auto_ptr的做法也是合理的：确保拥有权能够转移 。

如果auto_ptr的拷贝构造函数和赋值操作符的参数是auto_ptr const & ，那么实参的拥有权将不能转移。因为转移拥有权需要修改auto_ptr的成员变量，而实参确是一个const对象，不允许修改。

右值与**const &**

假设我们想写出下面的代码：

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

int main(int argc, char **argv) {
    auto_ptr<int> ptr1(auto_ptr<int>(new int(1)));  //使用临时对象进行拷贝构造
    auto_ptr<int> ptr2(NULL);
    ptr2 = (auto_ptr<int>(new int(2)));     //使用临时对象进行赋值
}
```

假设没有定义auto_ptr_ref类及相关的函数，那么这段代码将不能通过编译。主要的原因是，拷贝构造函数及赋值操作符的参数：auto_ptr<int>(new int(1))和 auto_ptr<int>(new int(2)) 都是临时对象 。临时对象属于典型的右值 ，而非const &是不能指向右值的(就是说给引用赋值) （参见More Effective C++，Item 19）。auto_ptr的拷贝构造函数及赋值操作符的参数类型恰恰是auto_ptr&，明显 非const &。

左值和右值

左值可以出现在赋值语句的左边或右边

右值只能出现在赋值语句的右边。

例如 x*y是个右值，编译表达式x*y=10;则出现错误

非**const**引用不能绑定右值

2011-9-3 16:44

提问者： 古刃霜剑 | 浏览次数：78次

为什么 const &a=40是对的，而&a=40是错的，其中具体的原理是怎样的？
&a是对a解引用了，是a的地址，当然不能把40复给他
而const必须在声明时初始化，也就是const &a=40是初始化而不是把40给赋值，所以编译能通过

同理，下面的两段代码，也不会通过编译：

```cpp
#include <iostream>
#include <memory>
using namespace std;
auto_ptr<int> f();
int main(int argc, char **argv) {
    auto_ptr<int> ptr3(f());  //使用临时对象进行拷贝构造
    auto_ptr<int> ptr4(NULL);
    ptr4 = f();       //使用临时对象进行赋值
}
```

```cpp
#include <iostream>
#include <memory>
using namespace std;
    auto_ptr<int> f(){
    return auto_ptr<int>(new int(3));  //这里其实也使用临时对象进行拷贝构造
}
```

普通类不会遇到这个问题，是因为他们的拷贝构造函数及赋值操作符（不管是用户定义还是编译器生成的版本），参数都是const &。

auto_ptr_ref之目的

传说当年C++标准委员会的好多国家，因为这个问题都想把auto_ptr从标准库中剔除。好在Bill Gibbons和Greg Colvin创造性地提出了auto_ptr_ref，解决了这一问题，世界清静了。

auto_ptr_ref之原理

很显然，下面的构造函数，是可以接收auto_ptr临时对象的。

```cpp
auto_ptr(auto_ptr __a) throw() : _M_ptr(__a.release()) { }
```

但另一个问题也很显然：上述构造函数不能通过编译。如果能通过编译，就会陷入循环调用。我们稍作修改：

```cpp
auto_ptr(auto_ptr_ref<element_type> __ref) throw()  //element_type就是auto_ptr的模板参数。

      : _M_ptr(__ref._M_ptr) { }
```

该版本的构造函数，可以接收auto_ptr_ref的临时对象。如果auto_ptr可以隐式转换到auto_ptr_ref，那么我们就能够用auto_ptr临时对象来调用该构造函数。这个隐式转换不难实现：

```cpp
template<typename _Tp1>

        operator auto_ptr_ref<_Tp1>() throw()

        { return auto_ptr_ref<_Tp1>(this->release()); }
```

至此，我们可以写出下面的代码，并可以通过编译：

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

int main(int argc, char **argv) {
    auto_ptr<int> ptr1(auto_ptr<int>(new int(1)));  //调用auto_ptr_ref版本的构造函数
}
```

同理，如果我们再提供下面的函数：

```cpp
auto_ptr&
operator=(auto_ptr_ref<element_type> __ref) throw()
{
    if (__ref._M_ptr != this->get())
    {
        delete _M_ptr;
        _M_ptr = __ref._M_ptr;
    }
    return *this;
}
```

那么，下面的代码也可以通过编译：

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

int main(int argc, char **argv) {
    auto_ptr<int> ptr2(NULL);
    ptr2 = (auto_ptr<int>(new int(2)));  //调用auto_ptr_ref版本的赋值操作符
}
```

auto_ptr_ref之本质

本质上，auto_ptr_ref赋予了auto_ptr“引用”的语义，这一点可以从auto_ptr_ref的注释看出：

```cpp
/**
   *  A wrapper class to provide auto_ptr with reference semantics.
```

```
 *  For example, an auto_ptr can be assigned (or constructed from)
 *  the result of a function which returns an auto_ptr by value.
 *
 *  All the auto_ptr_ref stuff should happen behind the scenes.
 */
template<typename _Tp1>
struct auto_ptr_ref
{
    _Tp1* _M_ptr;
    explicit
    auto_ptr_ref(_Tp1* __p): _M_ptr(__p) { }
};
```

auto_ptr_ref之代码

这里列出auto_ptr_ref相关的函数，共参考：

```
[cpp]
auto_ptr(auto_ptr_ref<element_type> __ref) throw()
: _M_ptr(__ref._M_ptr) {}

auto_ptr&
operator=(auto_ptr_ref<element_type> __ref) throw () {
    if (__ref._M_ptr != this->get()) {
        delete _M_ptr;
        _M_ptr = __ref._M_ptr;
    }
    return *this;
}

template<typename _Tp1>
operator auto_ptr_ref<_Tp1>() throw () {
    return auto_ptr_ref<_Tp1> (this->release());
}

template<typename _Tp1>
operator auto_ptr<_Tp1>() throw () {
    return auto_ptr<_Tp1> (this->release());
}
```

http://www.cnblogs.com/skyofbitbit/archive/2012/09/12/2681687.html

```

```

上一篇 使用隐式转换auto_ptr导致程序崩溃原因
下一篇 如何为排序，查找，set，map提供自定义比较功能

查看评论

1楼 zjufirefly 2014-05-03 17:00发表

explicit auto_ptr_ref(_Tp1* __p): _M_ptr(__p) { }是否需要实现为explicit?

发表评论

用户名： zjufirefly

评论内容：

提交

* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场

核心技术类目

全部主题 Hadoop AWS 移动游戏 Java Android iOS Swift 智能硬件 Docker OpenStack
VPN Spark ERP IE10 Eclipse CRM JavaScript 数据库 Ubuntu NFC WAP jQuery
BI HTML5 Spring Apache .NET API HTML SDK IIS Fedora XML LBS Unity
Splashtop UML components Windows Mobile Rails QEMU KDE Cassandra CloudStack
FTC coremail OPhone CouchBase 云计算 iOS6 Rackspace Web App SpringSide
Maemo Compuware 大数据 aptech Perl Tornado Ruby Hibernate ThinkPHP HBase
Pure Solr Angular Cloud Foundry Redis Scala Django Bootstrap