## 一、动态内存与智能指针

在C++中，动态内存通过一对运算符（new/delete）来申请和释放。但是这种机制存在问题（不容易确定何时释放内存，容易出现内存泄漏）。

为了更容易（同时也更安全）地使用动态内存，新的标准库提供了两种**智能指针**（smart point）类型来管理内存对象。智能指针的行为类似常规的指针，重要的区别是它们负责自动释放所指向的对象。新标准库提供的这两种智能指针的区别在于管理底层指针的方式：shared\_ptr允许多个指针指向同一个对象；unique\_ptr 则“独占”所指向的对象。标准库还定义了一个名为weak\_ptr的伴随类，它是一种若引用，指向shared\_ptr所管理的对象。这三种类型都定义在memory头文件中。

## shared\_ptr类：

类似vector，智能指针也是模板，因此，当我们创建一个智能指针时，必须提供额外的信息——指针可以指向的类型。与vector一样，我们在尖括号内给出类型，之后是所定义的这种智能指针的名字：

shared\_ptr<string> p1; //shared\_ptr，可以指向string。

shared\_ptr<list<int>> p2; //shared\_ptr，可以指向int的list。

默认初始化的智能指针中保存着一个空指针。

我们平时创建动态内存及释放内存的方式，

例如：

int \*pt = new int(11);

delete pt;

使用智能指针的方式是：

shared\_ptr<int> p = make\_shared<int>(11);

shared\_ptr允许多个指针指向同一个对象，即指向同一块内存。每个shared\_ptr都有一个关联的计数器，通常称为引用计数。无论何时我们拷贝一个shared\_ptr，计数器都会递增。例如，当用一个shared\_ptr初始化另一个shared\_ptr，或将它作为参数传递给一个函数以及作为函数返回值时，它所关联的计数器都会递增。当一个shared\_ptr被销毁（比如离开所属的作用域），其计数器就会减1。一旦一个shared\_ptr的计数器变为0，它就会自动释放自己所管理的对象。

|  |  |
| --- | --- |
| **shared\_ptr和unique\_ptr都支持的操作** | |
| shared\_ptr<T> sp  unique\_ptr<T> up | 空智能指针，可以指向类型为T的对象。 |
| p | 将p用作一个条件判断，若p指向一个对象，则为true。 |
| \*p | 解引用p，获得它指向的对象。 |
| p->mem | 等价于(\*p).mem。 |
| p.get() | 返回p中保存的指针。要小心使用，若智能指针释放了其对象，返回的指针所指向的对象也就消失了。 |
| swap(p, q)  p.swap(q) | 交换p和q中的指针。 |

|  |  |
| --- | --- |
| **shared\_ptr独有的操作** | |
| make\_shared<T>(args) | 返回一个shared\_ptr，指向一个动态分配的类型为T的对象。使用args初始化此对象。 |
| shared\_ptr<T>p(q) | P是shared\_ptr q的拷贝；此操作会递增q中的计数器。q中的指针必须能转化为T\*。 |
| P = q | P和q都是shared\_ptr，所保存的指针必须能相互转换。此操作会递减p的引用计数，递增q的引用计数；若p的引用计数变为0，则将其管理的原内存释放。 |
| p.unique() | 若p.use\_count() 为1，返回true；否则返回false |
| p.use\_count() | 返回与p共享对象的智能指针数量；可能很慢，主要用于调试。 |

mark\_shared函数

最安全的分配和使用动态内存的方式是调用一个名为mark\_shared的标准库函数。此函数在动态内存中分配一个对象并初始化它，返回指向此对象的shared\_ptr。与智能指针一样，mark\_shared也定义在memory中。

eg：

//指向一个值为42的int的shared\_ptr

shared\_ptr<int> p3 = mark\_shared<int>(42);

//p4指向一个值为“9999999999”的string

shared\_ptr<string> p4 = mark\_shared<string>(10, ‘9’);

//p5指向一个值初始化的int，即，值为0

shared\_ptr<int> p5 = mark\_shared<int>();

我们通常使用auto定义一个对象来保存mark\_shared的结果，这种方式比较简单。

//p6指向一个动态分配的空 vector<string>

auto p6 = mark\_shared<vector<string>>();

### 定义strBlob类：

使用动态内存的一个常见原因时允许多个对象共享相同的状态。书中定义了StrBlob类，引用了shared\_ptr的功能，实现我们所希望的数据共享。我们在StrBlob设置一个shared\_ptr来管理动态分配的vector。此shared\_ptr的成员将记录有多少个StrBlob共享相同的vector，并在vector的最后一个使用者被销毁时释放vector。

StrBlob类代码见src目录中strBlob项目，但比书中多增加了两个成员函数，printBlob()用于输出vector容器的内容，useCount()用于统计共享内存的引用计数。

**个人观点：采用了shared\_ptr的性质“允许多个指针指向同一个对象”，strBlob类的私有变量为shared\_ptr类型的指针，指向一个vector的数据变量，当该变量被实例化后，无论构造多少个strBlob类，都是指向那个之前构造好的vector数据变量，当销毁strBlob的对象是，对应的只会减少指向vector数据变量的指针，并不会销毁vector数据变量（在vector引用的指针数大于1的情况下）。**

## 直接管理内存：

动态内存的管理非常容易出错

使用new和delete管理动态内存存在三个常见问题：

1．忘记delete内存。忘记释放动态内存会导致人们常说的“内存泄漏”问题，因为这种内存永远不可能被归还给自由空间了。查找内容泄漏错误是非常困难的，以为通常应用程序运行很长时间后，真正耗尽内存时，才能检测到这种错误。

2．使用已经释放掉的对象。通过在释放内存后将指针置为空，有时可以检测出这种错误。

3．同一块内存释放掉两次。当有两个指针指向相同的动态分配对象时，可能发生这种错误。如果对其中一个指针进行delete操作，对象的内存就被归还给自己空间了。如果我们随后又delete第二个指针，自由控件就有可能被破坏。

相对于查找和修改这些错误来说，制造出这些错误要简单的多。

**Best Practices：坚持只使用智能指针，就可以避免所有这些问题。对于一块内存，只有在没有任何智能指针指向它的情况下，智能指针才会自动释放它。**

## shared\_ptr和new结合使用：

接收指针参数的智能指针构造函数是explicit的。因此，我们不能将一个内置指针隐式转换为一个智能指针，必须使用直接初始化形式来初始化一个智能指针。

shared\_ptr<int> p1 = new int(1024); //错误：必须使用直接初始化形式。

shared\_ptr<int> p2(new int(1024)); //正确：使用了直接初始化形式。

|  |  |
| --- | --- |
| **定义和改变shared\_ptr的其他方法** | |
| shared\_ptr<T> p(q) | P管理内置指针q所指向的对象；q必须指向new分配的内存，且能够转换为T\*类型。 |
| shared\_ptr<T> p(u) | p从unique\_ptr u那里接管了对象的所有权；将u置为空 |
| shared\_ptr<T> p(q, d) | p接管了内置指针q所指向的对象的所有权。q必须能转换为T\*的类型。p将使用可调用对象d来代替delete。 |
| shared\_ptr<T> p(p2, d) | 如前表所示，p是shared\_ptr p2的拷贝，唯一的区别是p将用于可调用对象d来代替delete。 |
| p.reset()  p.reset(q)  p.reset(q, d) | 若p是唯一指向其对象的shared\_ptr，reset会释放此对象。若传递了可选的参数内置指针q，会令p指向q，否则会将p置为空。若还传递了参数d，将会调用d而不是delete来释放q。 |

## 智能指针和异常：

**注意：**

智能指针可以提供对动态分配的内存安全而又方便的管理，但这建立在正确使用的前提下。为了正确使用智能指针，我们必须坚持一些基本规范。

1. 不使用相同的内置指针值初始化（或reset）多个智能指针。
2. 不delete get()返回的指针。
3. 不使用get()初始化或reset另一个智能指针。
4. 如果你使用get()返回的指针，记住当最后一个对应的智能指针销毁后，你的指针就变为无效了。
5. 如果你使用智能指针管理的资源不是new分配的内存，记住传递给它一个删除器。

## unique\_ptr类：

一个unique\_ptr“拥有”它所指向的对象。与shared\_ptr不同，某个时刻只能有一个unique\_ptr指向一个给定的对象。当unique\_ptr被销毁时，它所指向的对象也被销毁。

|  |  |
| --- | --- |
| **unique\_ptr操作** | |
| unique\_ptr<T> u1  unique\_ptr<T, D> u2 | 空unique\_ptr，可以指向类型为T的对象。u1会使用delete来释放它的指针；u2会使用一个类型为D的可调用的对象来释放它的指针。 |
| unique\_ptr<T, D> u(d) | 空unique\_ptr，指向类型为T的对象，用类型为D的对象d代替delete。 |
| u = nullptr | 释放u指向的对象，将u置为空。 |
| u.release() | u放弃对指针的控制权，返回指针，并将u置为空。 |
| u.reset() | 释放u指向的对象。 |
| u.reset(q)  u.reset(nullptr) | 如果提供了内置指针q，令u指向这个对象；否则将u置为空。 |

## weak\_ptr类：

weak\_ptr是一种不控制所指向对象生存期的智能指针，它指向由一个shared\_ptr管理的对象。将一个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。

|  |  |
| --- | --- |
| **weak\_ptr** | |
| weak\_ptr<T> w | 空weak\_ptr可以指向类型为T的对象。 |
| weak\_ptr<T> w(sp) | 与shared\_ptr sp指向相同对象的weak\_ptr。T必须能转换为sp指向的类型。 |
| w = p | p可以是一个shared\_ptr或一个weak\_ptr。赋值后与w与p共享对象。 |
| w.reset() | 将w置为空。 |
| w.use\_count() | 与w共享对象的shared\_ptr的数量。 |
| w.expired() | 若w.use\_count()为0，返回true，否则返回false。 |
| w.lock() | 如果expired为true，返回一个空shared\_ptr；否则返回一个指向w的对象的shared\_ptr。 |

当我们创建一个weak\_ptr时，要用一个shared\_ptr来初始化它：

auto p = make\_shared<int> (42);

weak\_ptr<int> wp(p); //wp弱共享p；p的引用技术未改变。