12 动态内存

1，动态内存与智能指针

静态内存用来保存局部static对象，类static数据成员以及定义在任何函数外的变量。static对象在使用之前分配，在程序结束时销毁。

栈内存用来保存定义在函数内的非static对象。分配在静态或栈内存中的对象由编译器自动创建和销毁。

除了静态内存和栈内存，每个程序还拥有一个内存池，这部分内存被称作自由空间或堆。

程序用堆来存储动态分配的对象，动态对象的生存期由程序来控制，当动态对象不再使用时，我们的代码必须显示地销毁它们。

动态内存的管理通过一对运算符来完成：new和delete。

新标准库提供的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。

weak\_ptr指向shared\_ptr所管理的对象。

参考文档：shared\_ptr和unique\_ptr都支持的操作

参考文档：shared\_ptr独有的操作

最安全的分配和使用动态内存的方法是调用一个名为make\_shared的标准库函数。

make\_shared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。

例子：shared\_ptr<int> p3 = make\_shared<int>(42);

当进行拷贝或赋值操作时，每个shared\_ptr都会记录有多少个其他shared\_ptr指向相同的对象。

auto p = make\_shared<int>(42); auto q(p);这时候对象有两个引用者

我们可以认为每个shared\_ptr都有一个关联的计数器，通常称其为引用计数。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。

当指向一个对象的最后一个shared\_ptr被销毁时，shared\_ptr类会自动销毁此对象。它是通过另一个特殊的成员函数--析构函数完成的。

shared\_ptr在无用之后仍然保留的一种可能情况是，你将shared\_ptr存放在一个容器中，随后重排了容器，从而不再需要某些元素，这种情况下，你应该确保用erase删除不再需要的shared\_ptr元素。

程序使用动态内存出于以下三种原因之一：1，程序不知道自己需要使用多少对象；2，程序不知道所需对象的准确类型；3，程序需要在多个对象间共享数据

使用动态内存的一个常见原因是允许多个对象共享相同的状态。

C++定义了两个运算符来分配和释放动态内存，运算符new分配内存，delete释放内存。相对于智能指针，使用这两个运算符管理内存非常容易出错。

使用new动态分配：string \*ps = new string; string \*ps = new string(10,'9')

动态分配const对象const int \*pci = new const int(1024);

我们传递给delete的指针必须指向动态分配的内存，或者是一个空指针。释放一块非new分配的内存，或者将相同的指针值释放多次，其行为都是未定义的。

当指针离开其作用域时，他所指向的对象什么也不会发生。如果这个指针指向的是动态内存，那么内存将不会被自动释放。

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；2，使用已经释放掉的内存；3，同一块内存释放两次

坚持只使用智能指针就可以避免所有这些问题。

delete之后重置指针，这只是提供了有限的保护。

使用new创建智能指针

shared\_ptr<int> p1 = new int(1024);//错误 shared\_ptr<int> p2(new int(1024))//正确

不要混合使用普通指针和智能指针。也不要使用get初始化另一个智能指针或为智能指针赋值。

智能指针类型类型定义了一个名为get的函数，它返回一个内置指针，指向智能指针管理的对象。

当我们创建shared\_ptr时，可以传递一个（可选的）指向删除器函数的参数：shared\_ptr<connection> p(&c, end\_connection)

参考文档：智能指针陷阱

与shared\_ptr不同，某个时刻只能有一个unique\_ptr指向一个给定对象。当unique\_ptr被销毁时，它指向的对象也被销毁。

unique\_ptr不支持拷贝和赋值，但可以通过release或reset将指针所有权从一个unique\_ptr转移给另一个unique。

release返回的指针通常被用来初始化另一个智能指针或给另一个智能指针赋值。

向unique\_ptr传递删除器 unique\_ptr<connection, decltype(end\_connection)\*> p(&c,end\_connection)

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不会改变它的引用计数。

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。

参考文档：weak\_ptr的操作

2 动态数组

大多数应用应该使用标准库容器而不是动态分配的数组。使用容器更为简单，更不容易出现内存管理错误并且可能有更好性能。

int \*pia = new int[get\_size()];//get\_size必须返回整型，但不必是常量。

动态数组并不是数组类型。

释放动态数组 delete []pia

可以使用unique\_ptr管理动态数组：unque\_ptr<int[]> up(new int[10]); up.release()//自动用delete[]销毁指针

标准库allocator类帮助我们将内存分配和对象构造分离开来。

allocator<string> alloc;auto const p = alloc.allocate(n);//分配n个未初始化的string

参考文档：allocator类及其算法

allocator的construct函数接受一个指针和零个或多个额外参数，在给定位置构造一个元素。

为了使用allocate返回的内存，我们必须用construct构造对象。使用未构造的内存，其行为是未定义的。

当我们用完对象后，必须对每个构造的元素调用destory来销毁它们。

一旦元素被销毁后，就可以重新使用这部分内存来保存其他string，也可以将其归还给系统。释放内存通过deallocate完成。

参考文档：allocator算法

3 使用标准库：文本查询程序

编程