动态内存和智能指针

动态分配对象的生存期通过显式的分配（new）与释放（delete）来控制。但是靠程序员手动进行这些操作容易出问题，因为释放的时机错误或者可能忘记释放空间从而导致内存泄漏等问题。

C++11的标准库提供了两个智能指针类型帮助管理动态对象，智能指针和常规指针不同之处在于智能指针会负责自动地安全地释放指向的对象。

shared\_ptr允许多个指针指向同一个对象；unique\_ptr独占所指向的对象——其他指针无法指向该对象；weak\_ptr，与shared\_ptr一同使用，为弱引用，指向shared\_ptr管理的对象。三者都定义在头文件memory中。

ps：C++的智能指针使用引用计数法管理生命周期，weak\_ptr为弱引用不参与引用计数，用来解决shared\_ptr的循环引用问题。

关于内存空间：

1.静态内存用来保存全局变量与局部 static 对象，类 static 数据成员。即适用于静态和全局变量，生命周期长，但分配次数有限。

2.栈内存用来保存定义在函数内的非 static 对象。 即适用于局部变量，分配速度快但空间有限。

3.堆内存保存动态分配的对象。即适用于动态分配，空间大但需手动管理，易发生内存泄漏。

堆内存和栈内存的比较：

1.控制权：

栈由编译器自动分配和释放；堆由程序员分配和释放。

2.空间大小：

栈：windows下，栈是向低地址扩展的，是连续的内存区域，所以栈顶地址和栈的最大容量都是确定的，似乎一般是 2M 或 1M

堆：堆是向高地址扩展的，是不连续的内存区域。系统是用链表来存储空闲地址的。堆的大小由计算机的有效虚拟内存决定，因此空间大得多。

3.分配效率：栈上分配速度较快。堆上分配速度较慢，但使用方便。

4.系统响应：

栈：如果剩余空间不足，异常提示栈溢出（error:stack\_over\_flow）

堆：在记录空闲地址的链表中寻找空间大于所申请空间的堆结点，然后将该结点从空闲节点链表中删除。一般会在首地址处记录本次分配空间的大小。当堆空间不足时，会返回空指针（nullptr），提示内存分配失败。

5.存储内容：

栈：存储函数的各个参数、局部变量、函数返回地址等。第一个进栈的就是函数返回地址。

堆：内容由程序员决定。

6.生命周期：

栈：栈上的变量生命周期由函数的作用域决定，一旦函数结束，栈上的变量自动释放。

堆：堆上的变量生命周期由程序员控制，只有显式调用 delete 或 delete[] 时才会释放。

7.访问速度：

栈：由于栈是连续的内存区域，且由CPU直接支持，访问速度通常较快。

堆：堆是分散的内存区域，访问速度相对较慢，因为需要通过指针间接访问（在内存中跳转会出现命中失败所以会慢）。

8.线程安全：

栈：每个线程都有自己的栈，因此栈操作是线程安全的。

堆：堆是共享的，多个线程访问同一堆上的数据需要同步，避免数据竞争。

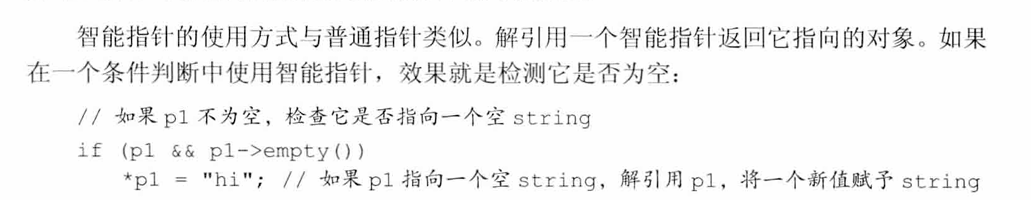
shared\_ptr类

shared\_ptr是模板，因此创建时必须提供指针可指向的类型。

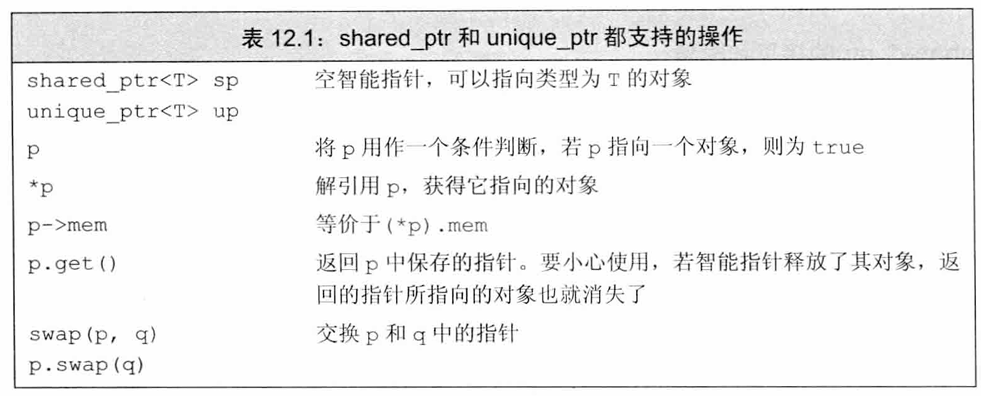
默认初始化的智能指针中保存着空指针。



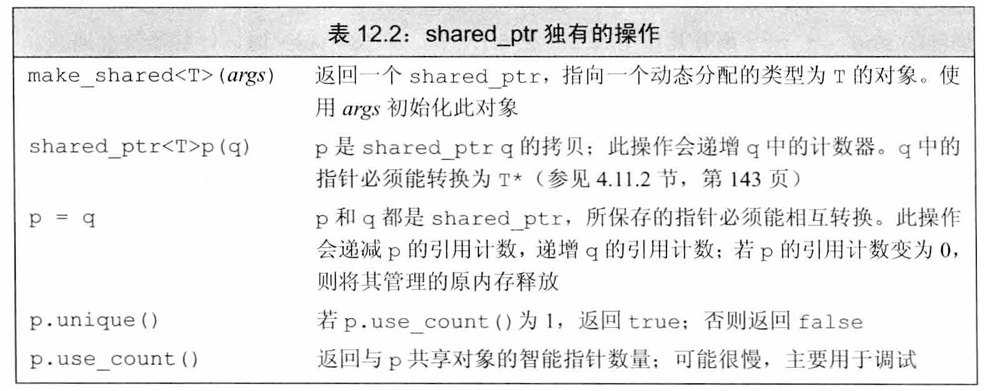
智能指针的使用方式和普通指针相似



shared\_ptr和unique\_ptr都支持的操作：



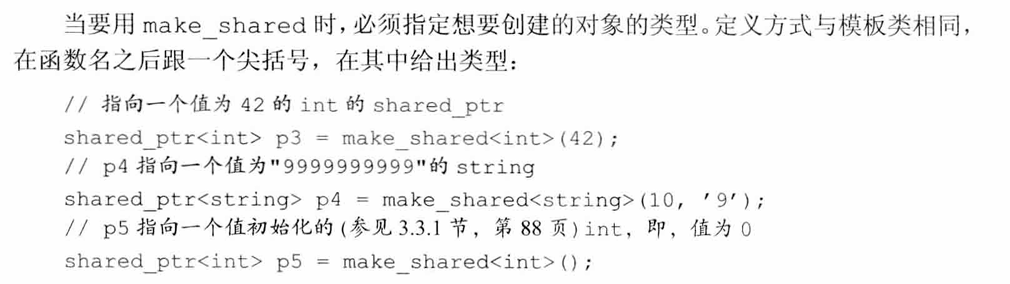
shared\_ptr独有的操作：



make\_shared<T>方法：

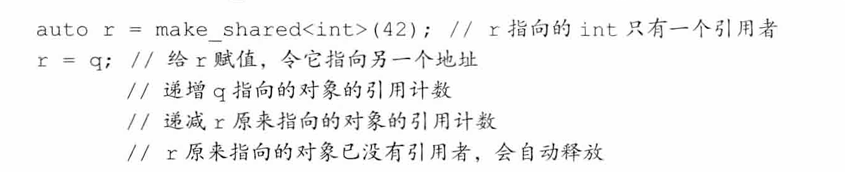
最安全的分配和使用动态内存的方法。

使用make\_shared<T>创建对象



shared\_ptr 的拷贝和赋值

每个 shared\_ptr 都有一个关联的计数器，如果拷贝一个 shared\_ptr，计数器就会递增（引用计数跟踪共享统一资源的shared\_ptr实例的数量）。如果 shared\_ptr 的计数器变为 0，就会自动释放管理的对象。



shared\_ptr自动销毁

shared\_ptr 通过析构函数来完成销毁（析构函数一般用来释放对象分配的资源）。当指向一个对象的所有shared\_ptr都被销毁后，shared\_ptr会自动销毁该对象。每次销毁shared\_ptr时析构函数会递减其指向对象的引用计数，如果计数变为 0，则销毁对象并释放内存。

需要保证shared\_ptr对象在无用之后不再保留，如果忘记销毁不需要的shared\_ptr实例，会浪费内存，一种常见的忘记销毁的情况是将 shared\_ptr 存放在一个容器中，此时引用计数不会归零。这时要注意用 erase 删除不需要的元素完成正常的内存释放。

使用动态内存的三种情况：

1.不知道需要使用多少对象。例如容器类。

2.不知道对象的准确类型。

3.需要在多个对象间共享内存。

如何使用动态内存在多个对象间共享内存

定义一个类，类的数据成员为一个 shared\_ptr。使用此 shared\_ptr 来管理一个 vector，即可实现在多个类对象间共享同一个 vector。当所有类对象都被销毁时 vector 才会被销毁。注意一个类只会与它的拷贝共享一个 vector，单独定义的两个类是不共享的。

直接管理内存

可以使用 new 和 delete 来直接管理内存。相比于智能指针，它们非常容易出错。

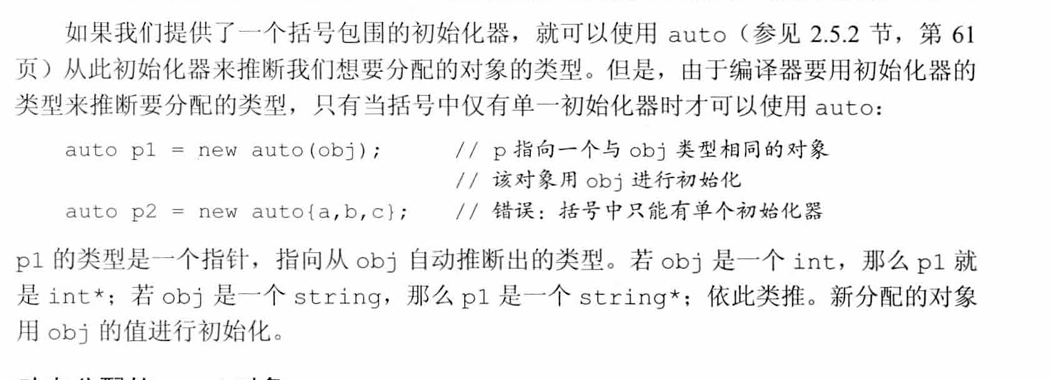
自己直接管理内存的类不能依赖默认合成的拷贝控制成员，通常都需要自己定义。而使用了智能指针的类则可以使用默认合成的版本。

使用new动态分配和初始化对象（new返回一个指针类型变量）

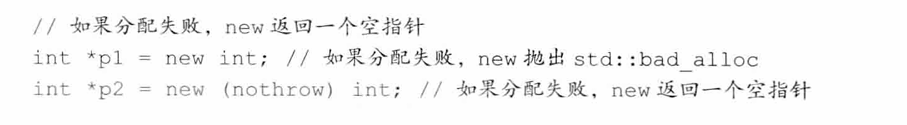


ps：对于类来说，值初始化与默认初始化没有什么区别，对于内置类型来说，值初始化对象会有一个良好的值，默认初始化对象值未定义。值初始化只需加括号即可。建议对动态分配对象进行值初始化。

可以让编译器推测分配对象的类型



如何阻止因内存耗尽无法分配对象抛出报错“bad\_alloc”



使用定位 new 可以向 new 传递参数，传递 nothrow 可以阻止 new 在分配失败的情况下抛出异常。

ps：bad\_alloc和nowhere可以阻止在行动失败

释放动态内存

使用 delete 表达式来释放动态内存，包括动态分配的 const 对象也是直接 delete 即可。

delete执行两个动作：

1.销毁指针所指对象（但没有销毁指针本身）

2.释放对应内存

释放一个非动态分配的指针或者相同的指针释放多次的行为都是未定义的。

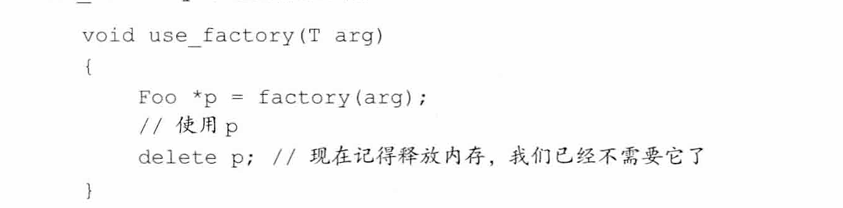
通常编译器不能分辨 delete 的对象是动态还是静态分配的对象，也不能分辨一个指针所指的内存是否已被释放。

动态对象直到被显式释放前都是存在的。书上下面这个例子：

在第一张图中，离开函数的作用域后指针p失效，但是指针p指向的对象没有被显式地释放，之后这块内存也就无法使用



要像这样显式释放指针p指向的对象



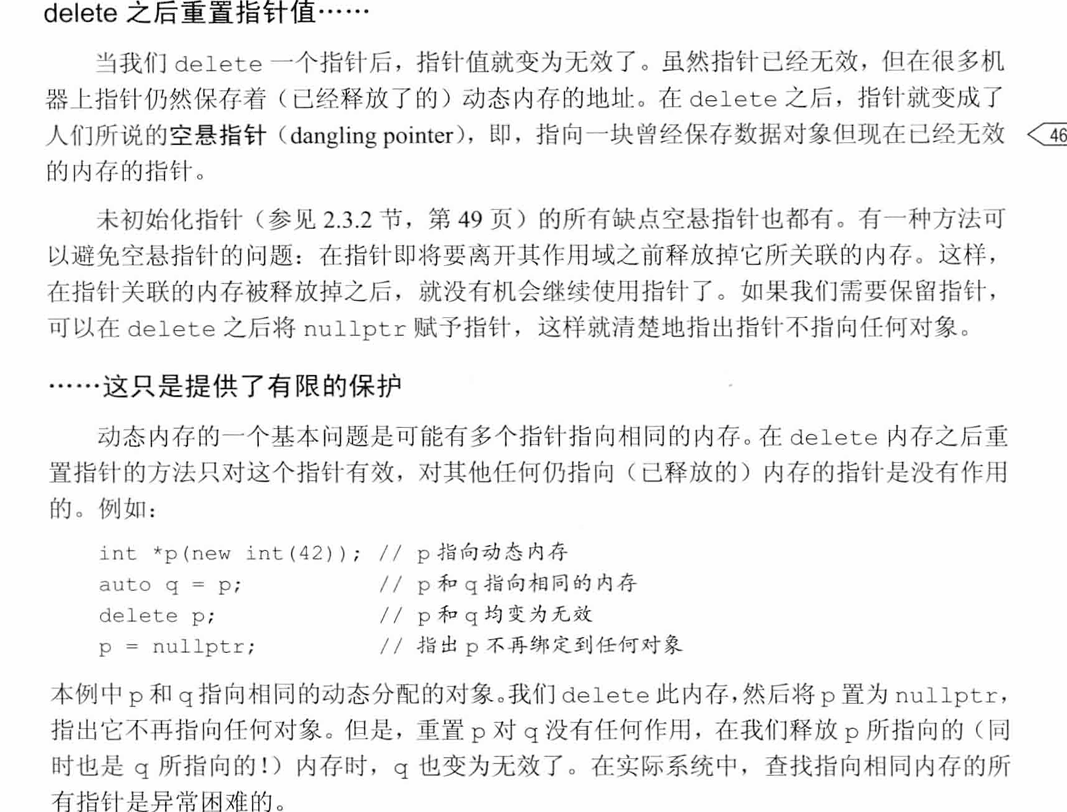
使用 new 和 delete 的三个常见错误：

1.忘记 delete 内存：内存泄漏。

2.使用已释放的的对象。

3.同一块内存释放两次

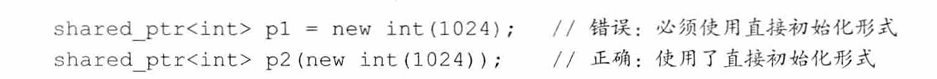
解决空悬指针问题的方法：



shared\_ptr和new结合使用

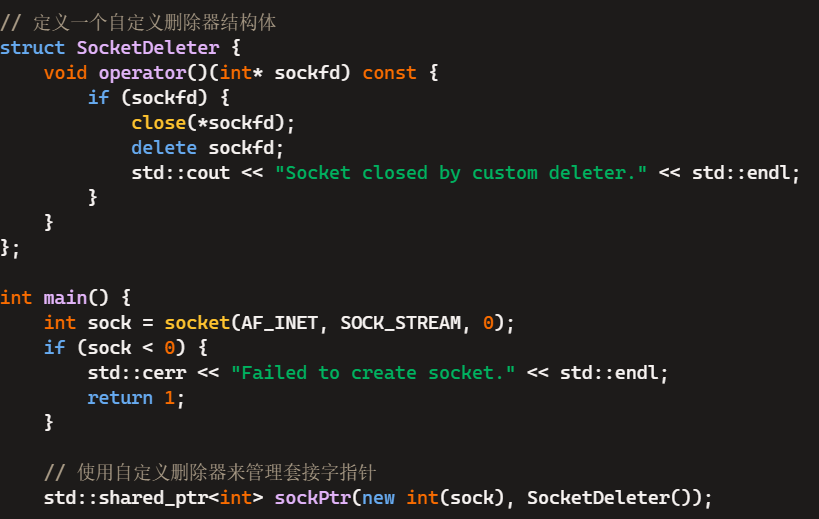
可以使用 new 初始化智能指针。但是最好还是用 make\_shared方法初始化智能指针。

接受指针参数的智能指针构造参数是 explicit 的，不能将内置指针隐式地转换为智能指针。因此不能使用赋值，只能用直接初始化。



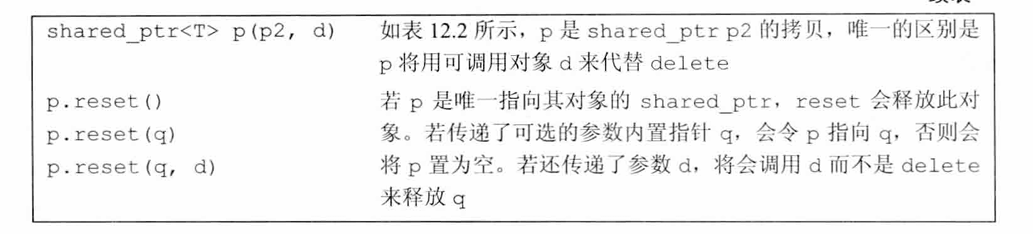
默认情况下用于初始化智能指针的普通指针只能指向动态内存，因为智能指针默认使用 delete 释放对象。如果将智能指针绑定到一个指向其他类型资源的指针上，要定义自己的方法代替 delete。

下面是一个从AI那得来的例子，定义删除器的结构体内部使用操作符重载了重载()操作符，在作用域结束时套接字会自动被智能指针关闭并释放内存



定义和改变shared\_ptr的其他方法：





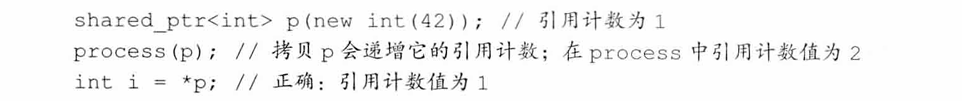
不要混用普通指针和智能指针

shared\_ptr 可以协调对象的析构，但仅限于自身的拷贝之间。这就是推荐使用 make\_shared 而不是 new 的原因。

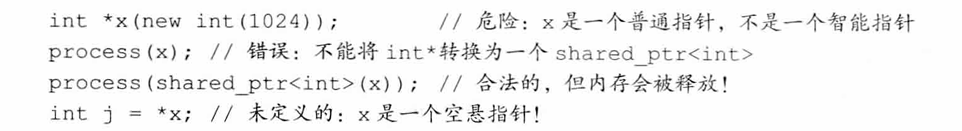
使用普通指针（即 new 返回的指针）来创建一个 shared\_ptr 有两个易错之处：

1.使用普通指针创建 shared\_ptr 后，又使用该普通指针访问动态对象。普通指针并不知道该对象何时被 shared\_ptr 所释放，随时可能变成空悬指针。

（1）因为传入的参数会被拷贝一份，在运行process函数时，p的引用计数一直为2，在离开函数的作用域之后引用计数也只会变为1，p指向的内存实际不会被释放。就像下面这个例子一样。



（2）如果创建一个内置指针，调用时依据这个内置指针创建一个智能指针临时变量，此时在表达式结束时引用计数会变为0，它指向的内存也会被释放，然后内置指针会变成一个空悬指针。



2.使用同一个普通指针创建了多个 shared\_ptr ，这就将同一块内存绑定到多个独立创建的 shared\_ptr 上了。

当将一个 shared\_ptr 绑定到一个普通指针后，就不要再用内置指针来访问所指内存了。

不要使用 get 初始化另一个智能指针或为智能指针赋值

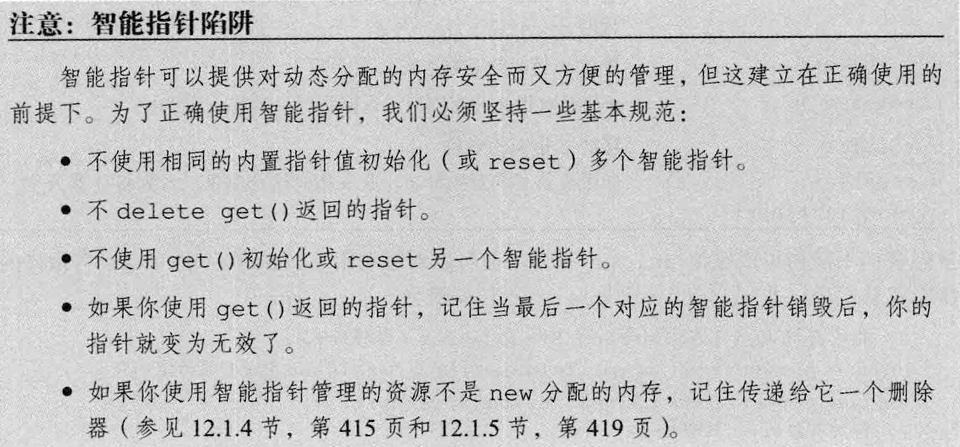
get方法的作用——智能指针的 get 函数返回一个内置指针。设计的目的是应对“需要向不能使用智能指针的代码传递一个内置指针”这种情况，且get返回的指针这个操作很危险，如果调用了delete释放内存，那么之后操作指针会出现未定义行为，且破坏了智能指针的内存管理。

shared\_ptr 的关联计数只应用于自己的拷贝，如果使用某智能指针的 get 函数初始化另一个智能指针，两个指针的计数是不关联的，销毁一个就会直接释放内存使另一个成为空悬指针。

智能指针和异常

如果使用智能指针时发生了异常，智能指针的内存会被释放掉；如果直接管理内存，在new和delete之间发生了异常，内存永远不会被释放。

智能指针使用规范：



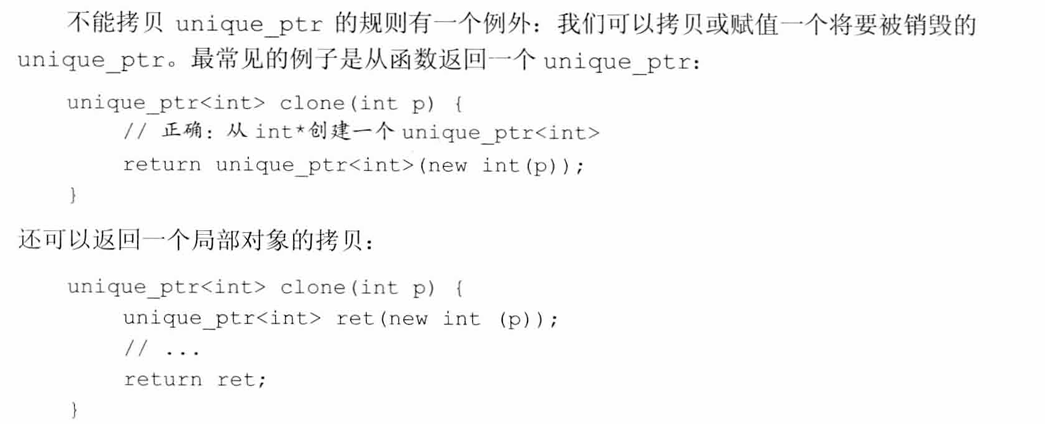
unique\_ptr类

同一时刻只能有一个 unique\_ptr 指向一个给定对象。当 unique\_ptr 被销毁时，指向对象也被销毁。

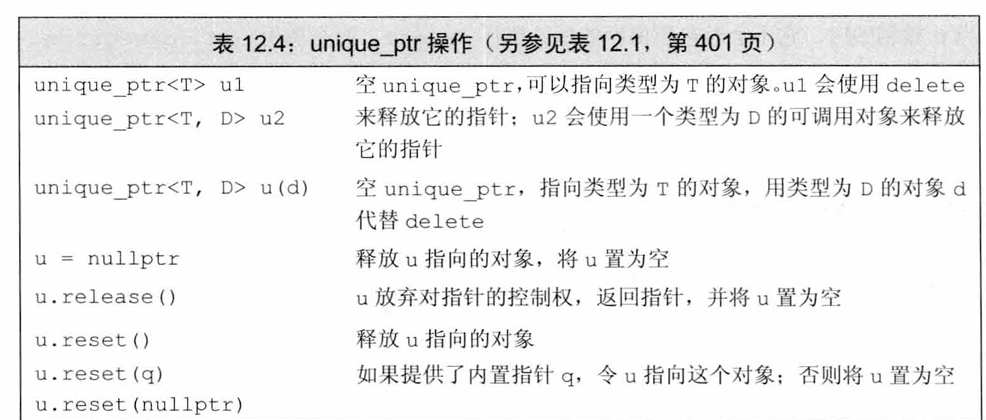
定义 unique\_ptr 时，需要绑定到一个 new 返回的指针上（不同于 shared\_ptr。因为unique\_ptr没有make\_shared)。

类似 shared\_ptr，初始化 unique\_ptr 必须采用直接初始化。（这里指使用 new 初始化）

因为 unique\_ptr 独有它指向的对象，所以不支持拷贝和赋值操作。（只有一种例外，返回一个局部的unique\_ptr拷贝或即将被销毁的unique\_ptr），例子如下：

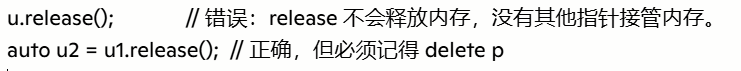


unique\_ptr类独有操作：



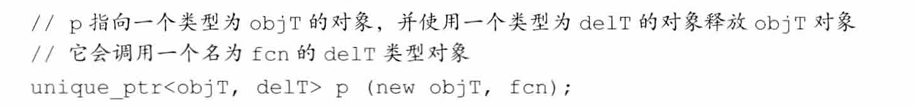
release方法的使用

release 返回的指针通常用来初始化其他智能指针或给其他智能指针赋值。且release 返回的指针不能空闲，必须有其他指针接管对象。如果是一个内置指针接管了 release 返回的指针，那么程序就要负责资源的释放。



向 unique\_ptr 传递删除器

类似 shared\_ptr，unique\_ptr 默认情况下使用 delete 释放它指向的对象。可以重载 unique\_ptr 中默认的删除器。但 unique\_ptr 管理删除器的方式与 shared\_ptr 不一样。unique\_ptr 将删除器的类型放在尖括号中，因为删除器会影响到unique\_ptr构造对象，而且在参数列表中也要提供删除器函数。



weak\_ptr类

weak\_ptr 指向一个由 shared\_ptr 管理的对象，它不控制所指向对象的生存期。

将一个 weak\_ptr 绑定到 shared\_ptr 不会改变 shared\_ptr 的引用计数。

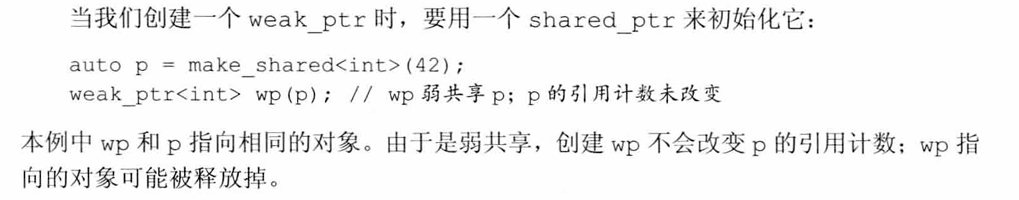
如果 shared\_ptr 都被销毁，即使还有 weak\_ptr 指向对象，对象依然会释放（因此不能使用 weak\_ptr 直接访问对象）。

weak\_ptr支持的操作：



ps：w.lock()方法返回的shared\_ptr会使得它指向对象的引用计数加一

weak\_ptr对象的初始化：



weak\_ptr 操作的应用：

image.png

动态数组

new 和 delete 运算符一次分配/释放一个对象。

C++ 中提供了两种一次分配一个对象数组的方法：

1.使用如 new int[10] 来分配一个对象数组；

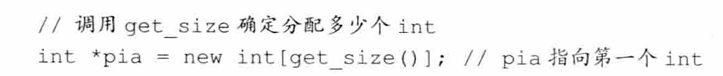
2.使用 allocator 类。allocator 类的优点是可以实现内存分配与对象构造的分离，更灵活地管理内存。

一般不需要使用动态分配的数组，而是使用如 vector 之类的 STL 容器。使用容器的类可以使用默认版本的拷贝、赋值、析构等操作，而分配动态数组的类必须定义自己版本的函数在相关操作时管理内存。

new和数组

使用方括号来分配一个对象数组，new 分配成功后返回指向第一个对象的指针。

方括号中的大小必须是整型，但不必是常量。

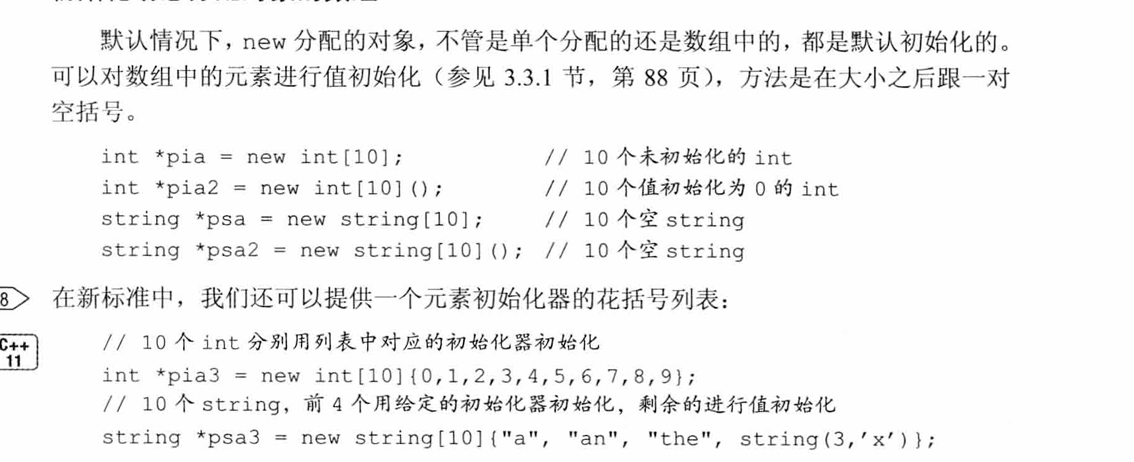


ps：“动态数组”并非真的数组，创建后得到的也不是一个数组类型对象，而是得到一个数组元素指针。同样使用类型别名定义动态数组也不会得到数组对象。因为不是数组类型，因此没有迭代器。因此也无法使用begin和end配合for循环处理元素。

初始化动态分配对象的数组

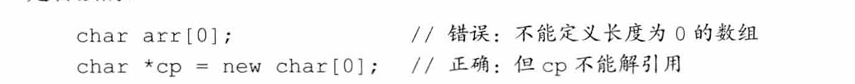
默认情况下 new 分配的对象不管是单个的还是动态数组，都是默认初始化的。但也可以进行值初始化：

下面图中除了pia、psa，其余都是值初始化



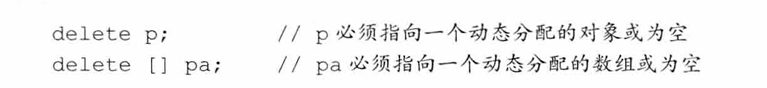
因为值初始化时不能提供参数，所以没有默认构造函数的类是无法动态分配数组的。

分配一个空的动态数组是合法的，new会返回一个合法的引用，但是只有使用new关键字才可以，直接默认初始化长度为0的数组是不可以的。



释放动态数组：

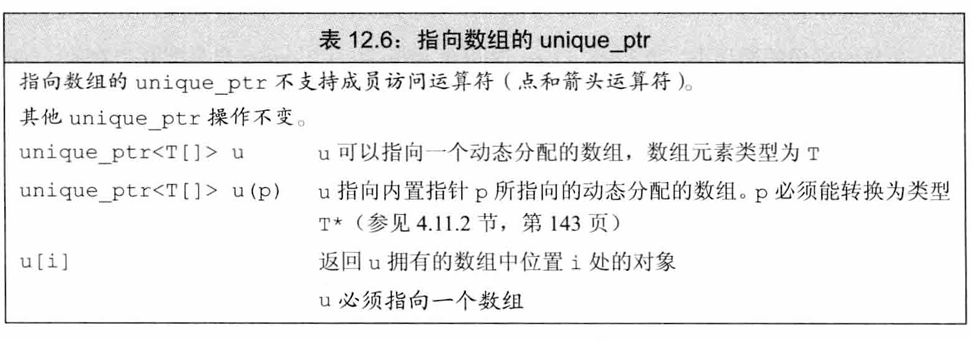
释放动态数组是必须要有[]操作符，如果没有就会报错未定义行为，



智能指针和动态数组——unique\_ptr

标准库提供了一个可以管理 new 分配的数组的 unique\_ptr 版本。这一版本的 unique\_ptr 自动使用 delete[] 来释放数组。

unique\_ptr对动态数组的操作：



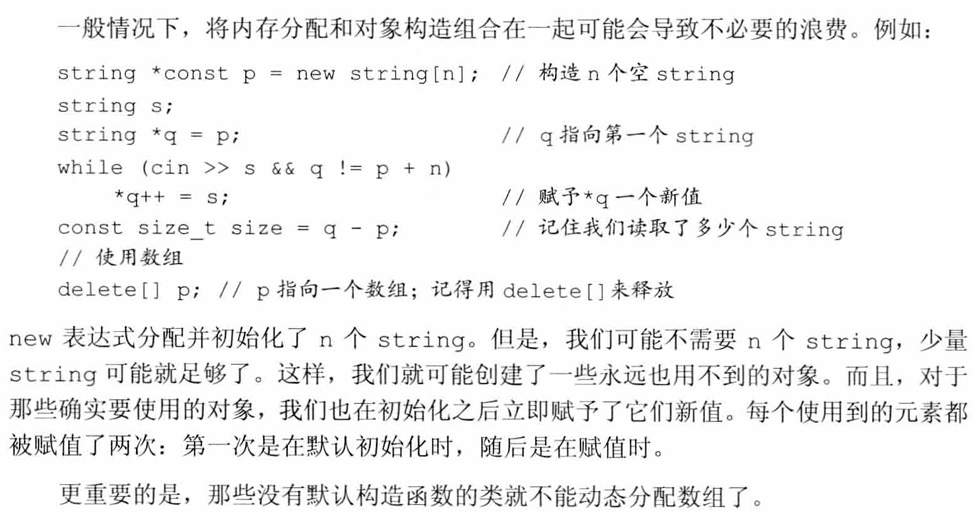
shared\_ptr不支持管理动态数组，如果一定要用shared\_ptr管理一个动态数组，必须自定义一个删除器。因为shared\_ptr释放内存用的是delete而非delete[]。

allocater类

标准库 allocator 类定义在头文件 memory 中，可以实现内存分配与对象构造的分离。

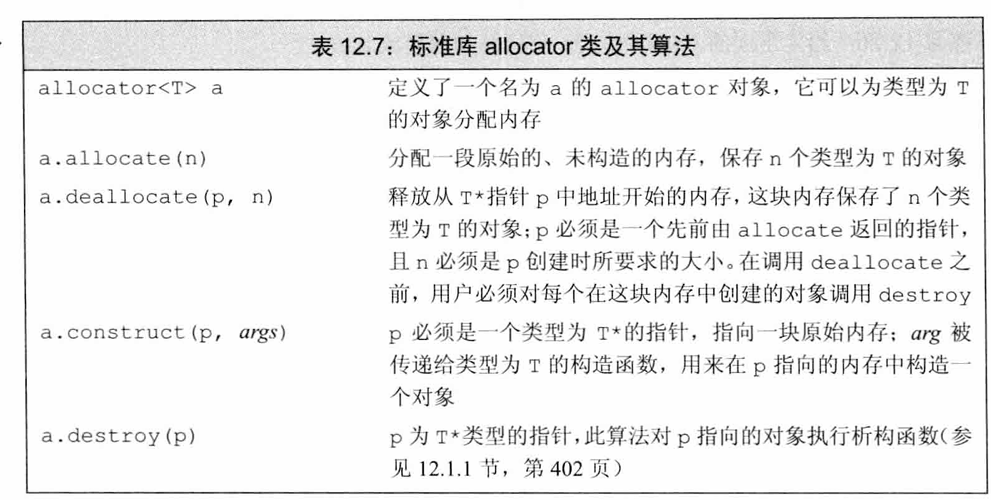
new 有一个局限性是它将内存分配和对象构造结合在了一起，对应的 delete 将对象析构和内存释放结合在了一起。

将内存分配和内存构造组合在一起会产生一些不必要的浪费：



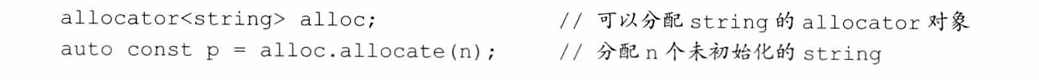
allocator 同样是一个类模板。

下面是allocator提供的方法：



ps：定义的 allocator 可以管理指定类型的内存分配、对象构造、对象销毁、内存释放四种操作，且这四种操作是分开的，分别对应一个函数。

allocator 分配未构造的内存



ps：返回值p指向分配的内存区域的首地址。

allocator 分配的内存是未构造的，需要使用 construct 成员函数按需在内存中构造对象。

construct 成员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，额外参数用来初始化构造的对象。





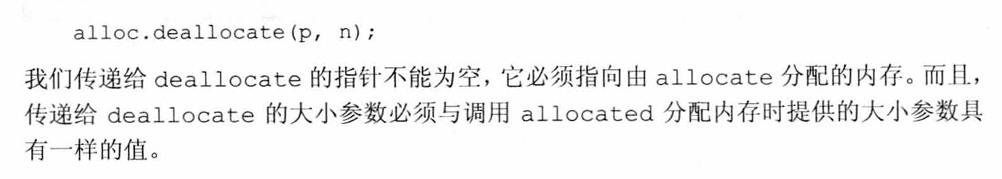
使用完对象后必须对每个被构造的元素调用destory销毁，destory 接受一个指针，对指向的对象执行析构函数。



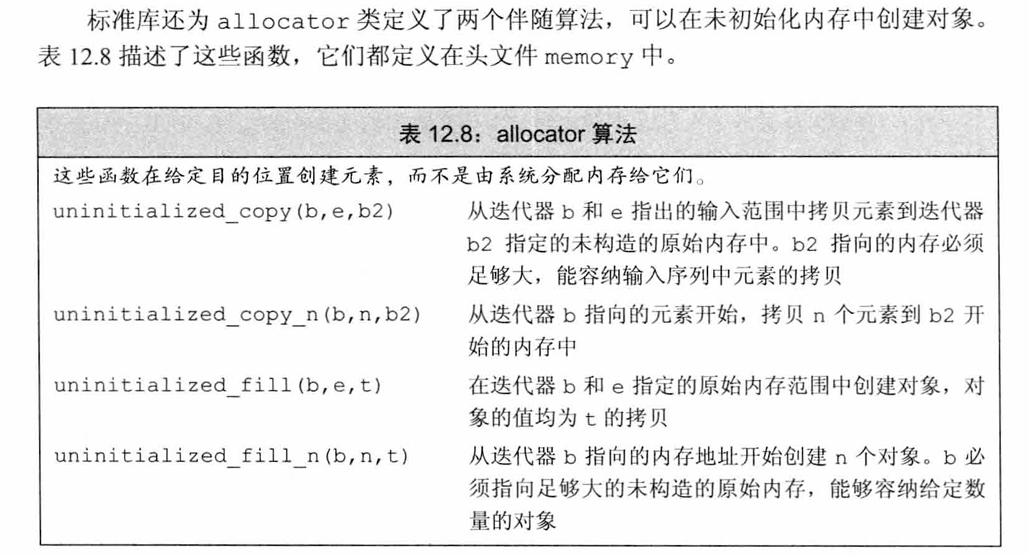
image.png

ps：construct 和 destory 一次都只能构造或销毁一个对象，要想完成对所有元素的操作，需要通过指针来遍历对每个元素进行操作。

在逐一销毁所有对象后释放内存空间



拷贝和填充未初始化内存的算法



书上给的使用的例子：

