动态分配对象的生存期与它们的创建位置无关;它们一直存在，直到被显式释放。

为了更安全地使用动态对象，库定义了两种智能指针类型来管理动态分配的对象。智能指针确保它们所指向的对象在适当的时候被自动释放。

我们的程序只使用静态或堆栈内存。静态内存用于本地静态对象（第6.1.1节），用于类静态数据成员（第7.6节），以及用于在任何函数外部定义的变量。堆栈内存用于函数内部定义的非静态对象。在静态或堆栈内存中分配的对象由编译器自动创建和销毁。堆栈对象只在定义它们的块执行时存在;静态对象在使用前被分配，当程序结束时被销毁。

* **静态内存**：用于全局变量、静态局部变量和类的静态成员，生命周期与程序一致。
* **栈内存**：用于局部变量，生命周期与代码块一致，由编译器自动管理。
* **堆内存**：用于动态分配的对象，生命周期由程序员控制，需要手动管理。

虽然有时候是必要的，但动态内存的正确管理是出了名的棘手。

在C++中，动态内存是通过一对操作符来管理的：new，它在动态内存中分配和可选地删除一个对象，并返回一个指向该对象的指针; delete，它获取一个指向动态对象的指针，销毁该对象，并释放相关的内存

shared\_ptr，允许多个指针引用同一个对象; unique\_ptr，“拥有”它所指向的对象。该库还定义了一个名为weak\_ptr的同伴类，它是对由shared\_ptr管理的对象的弱引用。所有这三个都在内存头中定义。

我们可以把shared\_ptr看作是一个关联的计数器，通常称为引用计数。

当我们给shared\_ptr分配一个新值时，当shared\_ptr本身被销毁时，比如当本地shared\_ptr超出作用域时，计数器会递减（第6.1.1节）。

是否使用计数器或其他数据结构来跟踪有多少个指针共享状态取决于实现。关键点在于，该类跟踪有多少shared\_ptrs指向同一个对象，并在适当的时候自动释放该对象。

如果您将shared\_ptrs放入容器中，并且随后需要使用其中的一些元素（而不是全部），请记住删除不再需要的元素。

程序倾向于使用动态存储器用于以下三个目的之一：

1.他们不知道需要多少个物体

2.他们不知道他们需要的对象的确切类型

3.他们希望在多个对象之间共享数据

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。

当我们描述这些操作符的工作方式时，会发现使用这些操作符管理内存比使用智能指针更容易出错。此外，与使用智能指针的类不同，管理自己内存的类不能依赖于复制、赋值和销毁类对象的成员的默认定义（第7.1.4节）。

出于与我们通常初始化变量相同的原因，初始化动态分配的对象也是一个好主意。

警告：管理动态内存容易出错使用new和delete来管理动态内存有三个常见的问题：1.忘记删除内存。忽略删除动态内存被称为“内存泄漏”，因为内存永远不会返回到空闲存储区。测试内存泄漏是很困难的，因为通常只有在应用程序运行足够长的时间以实际耗尽内存时才能检测到它们。

2.在对象被删除后使用它。这个错误有时可以通过在删除后使指针为空来检测。

3.两次删除相同的记忆。当两个指针指向同一个动态分配的对象时，可能会发生此错误。如果删除被应用于其中一个指针，则对象的内存被返回到空闲存储区。如果我们随后删除第二个指针，那么空闲存储可能会损坏。这类错误很容易犯，而不是发现和修复。

您可以通过专门使用智能指针来避免所有这些问题。只有当没有剩余的智能指针指向该内存时，智能指针才会负责删除该内存。

当我们删除一个指针时，这个指针就变得无效了。虽然指针无效，但在许多机器上，指针继续保存（释放的）动态内存的地址。删除后，指针变成所谓的悬挂指针。悬挂指针是指曾经保存对象但不再保存的内存。

动态内存的一个基本问题是，可能有多个指针指向同一个内存。重置用于删除该内存的指针可以让我们检查该特定指针，但对仍然指向（释放）内存的任何其他指针没有影响。

使用内置指针访问智能指针所拥有的对象是危险的，因为我们可能不知道该对象何时被销毁。

使用get only可将对指针的访问权传递给您知道不会删除该指针的代码。特别是，永远不要使用get来初始化或分配给另一个智能指针。

智能指针只有在正确使用时才能为处理动态分配的内存提供安全和方便。要正确使用智能指针，我们必须遵守一系列约定：

不要使用同一个内置指针值来初始化（或重置）多个智能指针。

不要删除get（）返回的指针

·不要使用get（）初始化或重置另一个智能指针。

·如果使用get（）返回的指针，请记住，当最后一个相应的智能指针消失时，指针将变得无效。

·如果你使用智能指针来管理new分配的内存以外的资源，记得传递一个删除器（第12.1.4节和第12.1.5节）。

weak\_ptr（表12 - 5）是一个智能指针，它不控制它所指向的对象的生存期。相反，weak\_ptr指向由shared\_ptr管理的对象。将weak\_ptr绑定到shared\_ptr不会更改该shared\_ptr的引用计数。一旦指向对象的最后一个shared\_ptr消失，对象本身将被删除。即使有weak\_ptr指向该对象，该对象也将被删除-因此命名为weak\_ptr，它捕获了weak\_ptr“弱”共享其对象的思想。

大多数应用程序应该使用库容器，而不是动态分配的数组。使用容器更容易，不太可能包含内存管理错误，并且可能提供更好的性能。

不要在类的代码中分配动态数组，直到你阅读了第13章。

重要的是要记住，我们所说的动态数组没有数组类型。

如果我们在删除指向数组的指针时忘记了括号，或者在删除指向对象的指针时使用了括号，编译器不太可能警告我们。相反，我们的程序在执行过程中很容易在没有警告的情况下行为不端。

**4. 实际应用场景**

解耦分配和构造的典型应用场景包括：

* **对象池（Object Pool）**：预先分配一块内存池，在需要时从池中取出内存并构造对象，使用完毕后将对象析构并归还内存池。
* **延迟初始化（Lazy Initialization）**：只有在第一次访问对象时才进行构造。
* **高性能系统**：在游戏引擎、实时系统等对性能要求较高的场景中，解耦分配和构造可以显著减少开销。

假设我们需要管理 100 个对象，但实际只使用其中的 10 个：

* **耦合方式**：一次性分配 100 个对象的内存并全部构造，浪费了 90 个对象的构造开销。
* **解耦方式**：先分配 100 个对象的内存，但只在需要时构造 10 个对象，其余内存保持未初始化状态，节省了资源。

在C++中，内存通过new表达式分配，通过delete表达式释放。该库还定义了一个分配器类，用于分配动态内存块。

分配动态内存的程序负责释放它们分配的内存。正确地释放动态内存是一个丰富的bug来源：要么内存永远不会被释放，要么在仍然有指针引用内存时被释放。新的库定义了智能指针shared\_ptr、unique\_ptr和weak\_ptr，它们使管理动态内存更加安全。一旦没有其他用户使用该内存，智能指针就会自动释放该内存。在可能的情况下，现代C++程序应该使用智能指针。

我们可以只破坏那些实际上被构建的元素。