Item 21:优先考虑使用std::make_unique和std::make_shared而非new

让我们先对`std::make_unique`和`std::make_shared`做个铺垫。`std::make_shared`是C++11标准的一部分，但很可惜的是，`std::make_unique`不是。它从C++14开始加入标准库。如果你在使用C++11，不用担心，一个基础版本的`std::make_unique`是很容易自己写出的，如下：

```
template<typename T, typename... Ts>
std::unique_ptr<T> make_unique(Ts&&... params)
{
    return std::unique_ptr<T>(new T(std::forward<Ts>(params)...));
}
```

正如你看到的，`make_unique`只是将它的参数完美转发到所要创建的对象的构造函数，从新产生的原始指针里面构造出`std::unique_ptr`，并返回这个`std::unique_ptr`。这种形式的函数不支持数组和自定义析构，但它给出了一个示范：只需一点努力就能写出你想要的`make_uniqe`函数。需要记住的是，不要把它放到std命名空间中，因为你可能并不希望在升级厂家编译器到符合C++14标准的时候产生冲突。

`std::make_unique`和`std::make_shared`有三个make functions中的两个：接收抽象参数，完美转发到构造函数去动态分配一个对象，然后返回这个指向这个对象的指针。第三个make function 是`std::allocate_shared.`它和`std::make_shared`一样，除了第一个参数是用来动态分配内存的对象。

即使是对使用和不使用make函数创建智能指针的最简单比较，也揭示了为什么最好使用这些函数的第一个原因。例如：

```
auto upw1(std::make_unique<Widget>());  // with make func
std::unique_ptr<Widget> upw2(new Widget);  // without make func
auto spw1(std::make_shared<Widget>());  // with make func
std::shared_ptr<Widget> spw2(new Widget);  // without make func
```

我高亮了区别：使用new的版本重复了类型，但是make function的版本没有。(译者注：这里高亮的是Widget，用new的声明语句需要写2遍Widget，make function只需要写一次) 重复写类型和软件工程里面一个关键原则相冲突：应该避免重复代码。源代码中的重复增加了编译的时间，会导致目标代码冗余，并且通常会让代码库使用更加困难。它经常演变成不一致的代码，而代码库中的不一致常常导致bug。此外，打两次字比一次更费力，而且谁不喜欢减少打字负担？

第二个使用make function的原因和异常安全有段。假设我们有个函数按照某种优先级处理Widget：

```
void processWidget(std::shared_ptr<Widget> spw, int priority);
```

根据值传递`std::shared_ptr`可能看起来很可疑，但是Item 41解释了，如果processWidget总是复制`std::shared_ptr`(例如，通过将其存储在已处理的Widget的数据结构中)，那么这可能是一个可复用的设计选择。

现在假设我们有一个函数来计算相关的优先级

`int computePriority();`

并且我们在调用processWidget时使用了new而不是std:: make_shared

```
processWidget(std::shared_ptr<Widget>(new Widget), computePriority()); //
potential resource leak!
```

如注释所说，这段代码可能在new Widget时发生泄露。为何？调用的代码和被调用的函数都用`std::shared_ptr`s,且`std::shared_ptr`s就是设计出来防止泄露的。它们会在最后一个`std::shared_ptr`销毁时自动释放所指向的内存。如果每个人在每个地方都用`std::shared_ptr`s,这段代码怎么会泄露呢?

答案和编译器将源码转换为目标代码有关。在运行时，一个函数的参数必须先被计算，才能被调用，所以在调用processWidget之前，必须执行以下操作，processWidget才开始执行:

- 表达式'new Widget'必须计算，例如,一个Widget对象必须在堆上被创建
- 负责管理new出来指针的`std::shared_ptr<Widget>`构造函数必须被执行
- computePriority()必须运行

编译器不需要按照执行顺序生成代码。“new Widget"必须在`std::shared_ptr`的构造函数被调用前执行，因为new出来的结果作为构造函数的参数，但compute Priority可能在这之前，之后，或者之间执行。也就是说，编译器可能按照这个执行顺序生成代码:

1. 执行new Widget
2. 执行computePriority
3. 运行`std::shared_ptr`构造函数

如果按照这样生成代码，并且在运行是computePriority产生了异常，那么第一步动态分配的Widget就会泄露。因为它永远都不会被第三步的`std::shared_ptr`所管理了。

使用`std::make_shared`可以防止这种问题。调用代码看起来像是这样:

```
processWidget(std::make_shared<Widget>(), computePriority());
```

在运行时，`std::make_shared`和computePriority会先被调用。如果是`std::make_shared`，在computePriority调用前，动态分配Widget的原始指针会安全的保存在作为返回值的`std::shared_ptr`中。如果compu tePriority生成一个异常，那么`std::shared_ptr`析构函数将确保管理的Widget被销毁。如果首先调用computePriority并产生一个异常，那么`std::make_shared`将不会被调用，因此也就不需要担心new Widget(会泄露)。

如果我们将`std::shared_ptr`,`std::make_shared`替换成std::unique_ptr,std::make_unique,同样的道理也适用。因此，在编写异常安全代码时，使用std::make_unique而不是new与使用`std::make_shared`同样重要。

`std::make_shared`的一个特性(与直接使用new相比)得到了效率提升。使用`std::make_shared`允许编译器生成更小，更快的代码，并使用更简洁的数据结构。考虑以下对new的直接使用:

```
std::shared_ptr<Widget> spw(new Widget);
```

显然，这段代码需要进行内存分配，但它实际上执行了两次。Item 19解释了每个`std::shared_ptr`指向一个控制块，其中包含被指向对象的引用计数。这个控制块的内存在`std::shared_ptr`构造函数中分配。因此，直接使用new需要为Widget分配一次内存，为控制块分配再分配一次内存。

如果使用`std::make_shared`代替: `auto spw = std::make_shared_ptr<Widget>();`一次分配足矣。这是因为`std::make_shared`分配一块内存，同时容纳了Widget对象和控制块。这种优化减少了程序的静态大小，因为代码只包含一个内存分配调用，并且它提高了可执行代码的速度，因为内存只分配一次。此外，使用`std::make_shared`避免了对控制块中的某些簿记信息的需要，潜在地减少了程序的总内存占用。

对于`std::make_shared`的效率分析同样适用于`std::allocate_shared`，因此`std::make_shared`的性能优势也扩展到了该函数。

更倾向于使用函数而不是直接使用new的争论非常激烈。尽管它们在软件工程、异常安全和效率方面具有优势，但本item的意见是，更倾向于使用make函数，而不是完全依赖于它们。这是因为有些情况下它们不能或不应该被使用。

例如，没有make函数允许指定定制的析构(见item18和19)，但是`std::unique_ptr`和`std::shared_ptr`有构造函数这么做。给Widget自定义一个析构:

```
auto widgetDeleter = [](Widget*){...};
```

使用new创建智能指针非常简单:

```
std::unique_ptr<Widget, decltype(widgetDeleter)>
 upw(new Widget, widgetDeleter);

std::shared_ptr<Widget> spw(new Widget, widgetDeleter);
```

对于make函数，没有办法做同样的事情。

make函数第二个限制来自于其单一概念的句法细节。Item7解释了，当构造函数重载，有`std::initializer_list`作为参数和不用其作为参数时，用大括号创建对象更倾向于使用`std::initializer_list`作为参数的构造函数，而用圆括号创建对象倾向于不用`std::initializer_list`作为参数的构造函数。make函数会将它们的参数完美转发给对象构造函数，但是它们是使用圆括号还是大括号？对某些类型，问题的答案会很不相同。例如，在这些调用中，

```
auto upv = std::make_unique<std::vector<int>>(10, 20);
auto spv = std::make_shared<std::vector<int>>(10, 20);
```

生成的智能指针是否指向带有10个元素的`std::vector`，每个元素值为20，或指向带有两个元素的`std::vector`，其中一个元素值10，另一个为20 ?或者结果是不确定的?

好消息是这并非不确定：两种调用都创建了10个元素，每个值为20.这意味着在make函数中，完美转发使用圆括号，而不是大括号。坏消息是如果你想用大括号初始化指向的对象，你必须直接使用new。使用make函数需要能够完美转发大括号初始化，但是，正如item31所说，大括号初始化无法完美转发。但是，item30介绍了一个变通的方法：使用auto类型推导从大括号初始化创建std::initializer_list对象(见Item 2)，然后将auto创建的对象传递给make函数。

```
// create std::initializer_list
auto initList = { 10, 20 };
// create std::vector using std::initializer_list ctor
auto spv = std::make_shared<std::vector<int>>(initList);
```

对于std::unique_ptr,只有这两种情景（定制删除和大括号初始化）使用make函数有点问题。对于`std::shared_ptr`和它的make函数，还有至少2个问题。都属于边界问题，但是一些开发者常碰到，你也可能是其中之一。

一些类重载了operator new和operator delete。这些函数的存在意味着对这些类型的对象的全局内存分配和释放是不合常规的。设计这种定制类往往只会精确的分配、释放对象的大小。例如，Widget类的operator new和operator delete只会处理sizeof(Widget)大小的内存块的分配和释放。这种常识不太适用于`std::shared_ptr`对定制化分配(通过std::allocate_shared)和释放(通过定制化deleters)，因为std::allocate_shared需要的内存总大小不等于动态分配的对象大小，还需要再加上控制块大小。因此，适用make函数去创建重载了operator new 和 operator delete类的对象是个典型的糟糕想法。

与直接使用new相比，`std::make_shared`在大小和速度上的优势源于`std::shared_ptr`的控制块与指向的对象放在同一块内存中。当对象的引用计数降为0，对象被销毁(析构函数被调用).但是，因为控制块和对象被放在同一块分配的内存块中，直到控制块的内存也被销毁，它占用的内存是不会被释放的。

正如我说，控制块除了引用计数，还包含簿记信息。引用计数追踪有多少`std::shared_ptr`s指向控制块，但控制块还有第二个计数，记录多少个std::weak_ptrs指向控制块。第二个引用计数就是weak count。当一个std::weak_ptr检测对象是否过期时(见item 19),它会检测指向的控制块中的引用计数(而不是weak count)。如果引用计数是0(即对象没有`std::shared_ptr`再指向它，已经被销毁了)，std::weak_ptr已经过期。否则就没过期。

只要std::weak_ptrs引用一个控制块(即weak count大于零)，该控制块必须继续存在。只要控制块存在，包含它的内存就必须保持分配。通过`std::shared_ptr` make函数分配的内存，直到最后一个`std::shared_ptr`和最后一个指向它的std::weak_ptr已被销毁，才会释放。

如果对象类型非常大，而且销毁最后一个`std::shared_ptr`和销毁最后一个std::weak_ptr之间的时间很长，那么在销毁对象和释放它所占用的内存之间可能会出现延迟。

```
class ReallyBigType { … };

// 通过std::make_shared创建一个大对象
auto pBigObj = std::make_shared<ReallyBigType>();

…           // 创建 std::shared_ptrs 和 std::weak_ptrs
            // 指向这个对象，使用它们

…           // 最后一个 std::shared_ptr 在这销毁，
            // 但 std::weak_ptrs 还在

…           // 在这个阶段，原来分配给大对象的内存还分配着

…           // 最后一个std::weak_ptr在这里销毁；
            // 控制块和对象的内存被释放
```

直接只用new，一旦最后一个`std::shared_ptr`被销毁，ReallyBigType对象的内存就会被释放：

```
class ReallyBigType { … };

//通过new创建特大对象
std::shared_ptr<ReallyBigType> pBigObj(new ReallyBigType);

…           // 像之前一样，创建 std::shared_ptrs 和 std::weak_ptrs
            // 指向这个对象，使用它们

…           // 最后一个 std::shared_ptr 在这销毁，
            // 但 std::weak_ptrs 还在

            // memory for object is deallocated

…           // 在这阶段，只有控制块的内存仍然保持分配

…           // 最后一个std::weak_ptr在这里销毁；
            // 控制块内存被释放
```

如果你发现自己处于不可能或不合适使用`std::make_shared`的情况下，你将想要保证自己不受我们之前看到的异常安全问题的影响。最好的方法是确保在直接使用new时，在一个不做其他事情的语句中，立即将结果传递到智能指针构造函数。这可以防止编译器生成的代码在使用new和调用管理新对象的智能指针的构造函数之间发生异常。

例如，考虑我们前面讨论过的processWidget函数，对其非异常安全调用的一个小修改。这一次，我们将指定一个自定义删除器:

```
void processWidget(std::shared_ptr<Widget> spw, int priority);
void cusDel(Widget *ptr); // 自定义删除器
```

这是非异常安全调用:

```
//和之前一样，潜在的内存泄露
processWidget(
    std::shared_ptr<Widget>(new Widget, cusDel),
    computePriority()
);
```

回想一下:如果computePriority在"new Widget"之后，而在`std::shared_ptr`构造函数之前调用，并且如果computePriority产生一个异常，那么动态分配的Widget将会泄漏。

这里使用自定义删除排除了对`std::make_shared`的使用，因此避免这个问题的方法是将Widget的分配和`std::shared_ptr`的构造放入它们自己的语句中，然后使用得到的`std::shared_ptr`调用processWidget。这是该技术的本质，不过，正如我们稍后将看到的，我们可以对其进行调整以提高其性能:

```
std::shared_ptr<Widget> spw(new Widget, cusDel);
processWidget(spw, computePriority());  // 正确，但是没优化，见下
```

这是可行的，因为`std::shared_ptr`假定了传递给它的构造函数的原始指针的所有权，即使构造函数产生了一个异常。此例中，如果spw的构造函数抛出异常(即无法为控制块动态分配内存)，仍然能够保证cusDel会在new Widget产生的指针上调用。

一个小小的性能问题是，在异常不安全调用中，我们将一个右值传递给processWidget

```
processWidget(
    std::shared_ptr<Widget>(new Widget, cusDel), // arg is rvalue
    computePriority()
);
```

但是在异常安全调用中，我们传递了左值

```
processWidget(spw, computePriority());  //spw是左值
```

因为processWidget的`std::shared_ptr`参数是传值，传右值给构造函数只需要move，而传递左值需要拷贝。对`std::shared_ptr`而言，这种区别是有意义的，因为拷贝`std::shared_ptr`需要对引用计数原子加，move则不需要对引用计数有操作。为了使异常安全代码达到异常不安全代码的性能水平，我们需要用std::move将spw转换为右值.

```
processWidget(std::move(spw), computePriority());
```

这很有趣，也值得了解，但通常是无关紧要的，因为您很少有理由不使用make函数。除非你有令人信服的理由这样做，否则你应该使用make函数。

记住：

- 和直接使用new相比，make函数消除了代码重复，提高了异常安全性。对于`std::make_shared`和`std::allocate_shared`,生成的代码更小更快。
- 不适合使用make函数的情况包括需要指定自定义删除器和希望用大括号初始化
- 对于`std::shared_ptr`s, make函数可能不被建议的其他情况包括
  (1)有自定义内存管理的类和
  (2)特别关注内存的系统，非常大的对象，以及`std::weak_ptr`s比对应的`std::shared_ptr`s活得更久