44 SMART POINTERS in C++

前面我们学到，`new`在堆上分配内存，需要`delete`来删除内存，因为它不会自动地释放内存。而*smart pointers* (智能指针)是自动化实现这一过程的方式。

当你调用`new`的时候，你不需要再调用`delete`。在很多使用智能指针的情况下你甚至不需要调用`new`。

智能指针本质上是一个原始指针的包装：
当你创建一个智能指针，它会调用`new`并为你分配内存，然后基于你使用的智能指针类型，这些内存会在某一时刻自动释放。

1. unique_ptr

*unique_ptr*是一个*scoped pointer* (作用域指针)，意味着这个指针超出作用域时，它会被销毁，然后调用`delete`。见上节课

为什么叫它*unique*指针呢?
是因为它必须是*唯一*的。你不能复制一个`unique_ptr`，因为如果你复制一个它，那么你会有两个`unique_ptr`指向同一块内存。如果其中一个死去，它会释放那段内存，也就是说指向同一块内存的另一个`unique_ptr`指向了已经被释放的内存。所以你不能复制它。

你不能像如下一样调用`unique_ptr`:

```C++
std::unique_ptr<Entity> entity = new Entity();
```

因为它的构造函数是*explicit*的，意味着你需要*显式*地调用构造函数。

正确做法:

```C++
std::unique_ptr<Entity> entity(new Entity()); // <模板参数>
entity->Print();
```

你可以像对待一个原始指针一样用箭头操作符调用它的函数。

一个**更好**的做法是:

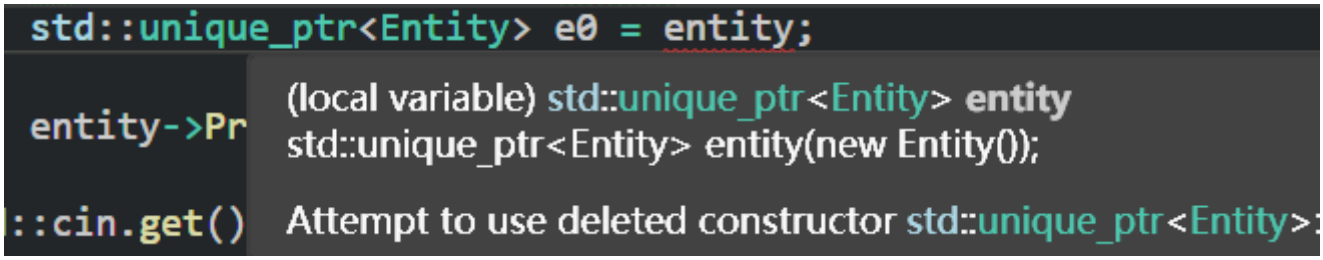
```C++
std::unique_ptr<Entity> entity = std::make_unique<Entity>();
```

这对于`unique_ptr`来说很重要，主要原因是出于*exception safety* （异常安全），如果构造函数碰巧抛出异常，它会稍微安全一些。你不会最终得到一个没有引用的*dangling pointer*（悬空指针）而造成过内泄漏。

如图，离开作用域的时候，我们的entity会自动销毁。

这是最简单的智能指针，非常有用且开销低（甚至没有开销），它只是一个*栈分配对象*，当栈分配对象死亡（销毁）时，它将在你的指针上调用`delete`，并释放内存。

前面提到了`unique_ptr`不能被复制。如果你去看它的定义，你会发现它的拷贝构造函数和拷贝构造操作符实际上被删除了，这就是为什么你运行如下代码时会编译错误。



这是专门用来防止你自掘坟墓的，因为你不能复制这个：其中一个指针死（失效）后，它们都会“死（失效）”，因为这个堆分配对象的底层内存会被释放。（意思就是说它其实不是指针而是个对象）

所以如果你想“分享”这个指针，这就是*shared pointer*（共享指针）的用处所在了。

2. shared_ptr

共享指针有点不同，它有点硬核，因为它还在底层做了很多其它的事情。
`shared_ptr`实现的方式实际上取决于编译器和你在编译器中使用的标准库。不过在作者所见过的所有系统中，它使用的都是*reference counting*（引用计数）.

引用计数基本上是一种方法，可以跟踪你的指针有多少个引用。一旦引用计数达到0，它就被删除了。

举个例子，我刚创建了一个共享指针，又创建了另一个共享指针来复制它，此时我的引用计数是2。第一个指针失效时，我的引用计数器减少1，然后最后一个失效时，我的引用计数回到0，就真的“dead”了，因此内存被释放。

这样可以通过编译（但不要这样使用！）：

```C++
std::shared_ptr<Entity> sharedEntity(new Entity());  //不要这样用!
```

不过你绝对不会想将`shared_ptr`这样用，在`unique_ptr`中，不直接调用`new`的原因是因为异常安全，但是在`shared_ptr`中有所不同。

因为`shared_ptr`需要分配另一块内存，叫做控制块，用来存储引用计数。如果你先创建一个new Entity，然后将其传递给`shared_ptr`的构造函数，它得做2次内存分配：先做一次new Entity的分配，然后是shared_ptr的控制内存块的分配。

然而如果你用`make_shared`，你能把它们组合起来：

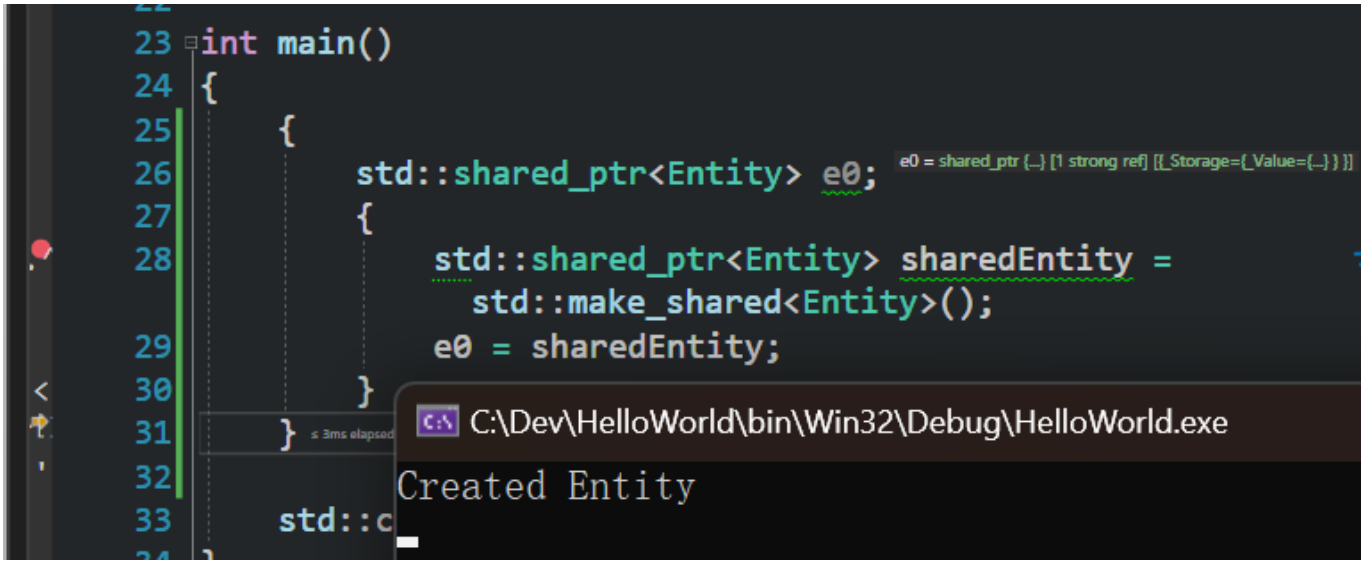
```C++
std::shared_ptr<Entity> sharedEntity = std::make_shared<Entity>();
```

这样更有效率，对于那些不喜欢用new、delete的人显然也是有帮助的。

共享指针的复制

有了共享指针，你当然可以进行复制。
下图代码中有两个作用域，可以看到里面这个作用域死亡时，这个sharedEntity失效了，然而并没有对Entity析构并删除，因为

e0仍然是有效的，并且持有对该Entity的引用。再按一下F10，当所有引用都没了，当所有追踪`shared_ptr`的栈分配对象都死亡后，底层的Entity才会从内存中释放并删除。



3. weak_ptr

还有一个东西你可以和`shared_ptr`一起使用

```C++
std::shared_ptr<Entity> sharedEntity = std::make_shared<Entity>();
std::weak_ptr<Entity> weakEntity = sharedEntity;
```

这里所做的，和之前复制sharedEntity所做的一样，但之前会增加引用计数而这里不会：
当你将一个`shared_ptr`赋值给另外一个`shared_ptr`，引用计数会增加。但是当你把一个`shared_ptr`赋值给一个`weak_ptr`时，它不会增加引用计数。

那这样的好处是什么？
如果你不想要Entity的所有权，就像你可能在排序一个Entity列表，你不关心它们是否有效，你只需要存储它们的一个引用就可以。
你可以知道它引用的底层对象还活着吗？如果是的话你就可以做其它操作。但是不会让底层对象保持存活，因为它不增加引用计数。

如上图，改成`weak_ptr`后，退出第一个作用域它就被销毁了。此时这个`weak_ptr`e0正指向一个无效的Entity。不过你可以问这个弱指针：你还有效吗?

这就是很有用的智能指针，但它们绝对没有完全取代`new`和`delete`关键字。只是当你要声明一个堆分配的对象而且不希望由自己来清理，这时候你就应该使用智能指针，尽量使用`unique_ptr`，因为它有较低的开销。但如果你需要在对象之间共享，不能使用`unique_ptr`的时候，就用`shared_ptr`