对象的构造和析构

一般而言，构造函数被安插在对象的定义处，而析构函数被安插在对象生命 周期结束前：

```
// Pseudo C++ Code  {
  Point point;
  // point.Point::Point() 一般被安插在这儿
  ...
  // point.Point::~Point() 一般被安插在这儿 }
```

当代码有一个以上的离开点的时候，析构函数则必须放在对象被构造之后的每 一个离开点之前。因此，尽可能将对象定义在接近要使用的地方，可以减少不 必要的构造对象和析构对象的代码被插入到自己的代码当中。

全局对象

一个全局对象，c++保证它在 `main()` 在第一次使用它之前将其构造，而在 `main()` 结束之前，将之析构掉。C 规定一个全局对象只能被一个常量表达式 (编译期可知)赋初值。而构造函数显然不是一个常量表达式。虽然全局对象在 编译期被即被置为 0，但真正的构造工作却需要直到程序激活后才能进行，而 这个过程就是所谓的静态初始化。我是这样理解，但我不保证正确，因为全局 变量，被放在 data segment (数据段)，data segment 是在编译期已经布置好 的，但构造函数的结果在编译期不能评估，因此先将对象的内容设置为 0，存 储在数据段，而等到程序激活时，这时候就可以通过构造函数对在数据段的全 局对象进行初始化了，而这就是所谓的静态初始化。

静态初始化的对象有一些缺点：如果构造函数支持异常机制，那么遗憾的是对 象的构造函数的调用，无法被放置与 try 块中，我们知道一个没有得到 catch 的 异常默认的调用 `terminate()` 函数。也就是说一个全局对象在构造过程中抛出 异常，将导致程序的终结，而更悲剧的是，你还无法来捕获并处理这个异常。 另一点在于，在不同文件中定义的全局变量，构造顺序有规则吗？我不知道。 即使有规则，如果不同的构造顺序对程序有影响的话，那么有多琐碎复杂...

Lippman 甚至建议：根本就不要使用那些需要静态初始化的全局对象。真的非要 一个全局对象，而且这个对象还需要静态初始化？那么我的方法是，用一个函 数封装一个静态局部对象，也是一样的效果嘛。

局部静态对象(Local Static Object)

下面一段代码：

:::C++ const Matrix& identity() {
static Matrix mat_identity;
// ...
return mat_identity;
}

因为静态语意保证了 mat_identity 在整个程序周期都存在，而不会在函数 `identity()` 退出时被析构，所以：

- mat_identity 的构造函数只能被施行一次，虽然 identity()可以被调用 多次。
- mat_identity 的析构函数只能被施行一次，虽然 identity()可以被调用 多次。

那么 mat_identity 的构造函数和析构函数到底在什么时候被调用？答案是:
mat_identity 的构造函数只有在第一次被调用时在被施行，而在整个程序退出 之时按构造相反的顺序析构局部静态对象。

对象数组(Array of Objects)

对于定义一个普通的数组，例如：

```
Point knots[ 10 ];
```

实际上背后做的工作则是：

1. 分配充足的内存以存储 10 个 Point 元素；
2. 为每个 Point 元素调用它们的默认构造函数(如果有的话，且不论是合成的还 是显式定义的)。编译器一般以一个或多个函数来完成这个任务。当数组的 生命周期结束的时候，则要逐一调用析构函数，然后回收内存，编译器同样 一个或多个函数来完成任务。这些函数完成什么功能，大概都能猜得出来。 而关于细节，不必要死扣了，每个编译器肯定都有些许差别。

参考：Lippman 的两本书《深度探索 C++对象模型》和《C++ Primer》。