如何形象地认识 RAII

如果把内存比作电脑 CPU 的草稿纸，所有临时的运算将都发生在这张大小有限的草稿纸上。【用 malloc 在堆上申请内存】这个行为就好比在草稿纸上画了一个方框，预约草稿纸上这块区域要用于未来的某种计算。这块区域将无法被挪作他用，直到这样的预约在某处被【注销】。

一个复杂的程序当中，将会有很多个体来进行这样的预约，随后在草稿纸的相应位置开始计算。为了使得，草稿纸不会趋于越来越满，每个预约人（往往对应一个对象）必须要记得在完成计算任务以后【注销】这次预约，把这块打过草稿的地方擦干净给别人用。但是预约人是有可能猝死的（异常/early return），一旦猝死，草稿纸上就少了一块空间。例如下面这个例子：

```
void someFunc(bool early_return) {
  int* a = (int*)malloc(5 * sizeof(int));

  if (early_return){
    return; // Oops, `a` dies before it can free anything!
  }

  free(a); // In some cases, this line won't run.
} // Outside this '{}' pair, 'a' is by no means accessible.
// But the heap memory it possesses stay unreleased.
```

这个例子中，变量 a 在离开作用域后就【死亡】了，但是它向内存预约并借用的空间却没有被正确释放，这就有点像【一笔坏账】。这样的行为便是构成【内存泄漏】最常见也是最本质的要素。为了机制性地（而不是依赖一个足够专注且聪明的神明，即程序员）来防止这种猝死带来的坏账堆积，我们就开始思考，究竟什么东西能够在预约人猝死的情况下依然完成【注销】的遗志？

换句话说，我们有一个【任何情况下都应该被完成的任务】，即注销与释放；同时我们又学习到，C++中也存在一个【任何情况下都会被调用的函数】，也就是对象的析构函数。这二者简直是天作之合！

没错，析构函数很类似于变量的【遗嘱】，它总是会在这个变量生命消亡的时候被调用（这个事实受到 C++标准的保护，所以不会随着环境发生改变）！所以很自然地出现一种编程原则（RAII）规定所有需要在堆上预约草稿纸空间的个体，都必须事先通过析构函数写好遗嘱，这样不管是病死老死车祸死，总是有人根据这个遗嘱来帮他注销预约过的空间。

这样一来，所有复杂个体的资源申请和释放，都被对偶地封装在了它的构造和析构的过程中，从而实现了（对于外部用户而言的）自动管理和保护。这就是 RAII 的本质。