对C 程序内存管理的精雕细琢 PDF转换可能丢失图片或格式，建议阅读原文 https://www.100test.com/kao_ti2020/134/2021_2022__E5_AF_B9C___E7_A8_8B_E5_c97_134367.htm 应用程序分配内存的方法，对程序的执行性能有着深刻的影响。目前，通用的内存分配方法本质上已非常高效，但仍有改进的空间。 内存分配，不可一层不变 今天，对绝大多数程序来说，通用的内存分配方法--此处指代分配算符（Allocator：即malloc或new），已达到了理想的速度及满足了低碎片率的要求，然而，在内存分配领域，一丁点的信息都值得探讨很久，某些特定程序关于分配模式的信息，将有助于实现专门的分配算符，可显著地提高大多数高性能要求程序的性能底线。有时，当通用内存分配算符平均耗费几百个时钟周期时，一个良好的自定义内存分配算符可能只需要不到半打的周期。 这就是为什么大多数高性能、高要求的应用程序（如GCC、Apache、Microsoft SQL Server），都有着它们自己的内存分配算符。也许，把这些专门的内存分配算符归纳起来，放进一个库中，是个不错的想法，但是，你的程序可能有不同的分配模式，其需要另外的内存分配算符，那怎么办呢？ 等等，还有呢，如果我们设计了一种特殊用途的内存分配算符，就可以不断发展下去，由此可从中筛选出一些，来组成一个通用目的的内存分配算符，如果此通用分配算符优于现有的通用分配算符，那么此项设计就是有效及实用的。 下面的示例使用了Emery小组的库--HeapLayers（http://heaplayers.org/），为了定义可配置的分配算符，其中使用了mixins（在C 社区中，也被称为Coplien递归模式）：通过参数化的基来定义类，每一层中

只定义两个成员函数，malloc和free：template struct Allocator : public T { void * malloc(size_t sz). void free(void* p). //系统相关的值 enum { Alignment = sizeof(double) }. //可选接口e size_t getSize(const void* p).}. 在每一层的实现中，都有可能向它的基类请求内存，一般来说，一个不依赖于外界的内存分配算符，都会处在层次的顶层--直接向前请求系统的new和0delete操作符、malloc和free函数。在HeapLayers的术语中，没有顶层堆，以下是示例：struct MallocHeap { void * malloc(size_t sz) { return std::malloc(sz). } void free(void* p) { return std::free(p). }}. 为获取内存，顶层堆也能通过系统调用来实现，如Unix的sbrk或mmap。getSize函数的情况就比较特殊，不是每个人都需要它，定义它只是一个可选项。但如果定义了它，你所需做的只是插入一个存储内存块大小的层，并提供getSize函数，见例1： 例1：template class SizeHeap { union freeObject { size_t sz. double _dummy. //对齐所需 }.public: void * malloc(const size_t sz) { //添加必要的空间 freeObject * ptr = (freeObject *)SuperHeap::malloc(sz sizeof(freeObject)). //存储请求的大小 ptr->sz = sz. return ptr 1. } void free(void * ptr) { SuperHeap::free((freeObject *) ptr - 1). } static size_t getSize (const void * ptr) { return ((freeObject *)ptr - 1)->sz. }}. SizeHeap是怎样实现一个实用的层，并挂钩于它基类的malloc与free函数的最好示例，它在完成一些额外的工作之后，把修改好的结果返回给使用者。SizeHeap为存储内存块大小，分配了额外的内存，再加上适当的小心调整（指union），尽可能地避免了内存数据对齐问题。不难想像，我们可构建一个debug堆，其通过特定模式在内存块之前或之后填充了一些字节，通过检查

是否模式已被保留，来确认内存的溢出。事实上，这正是HeapLayers的DebugHeap层所做的，非常的简洁。 