Modern C++ Camp

现代C++系统研发骨干特训营

李建忠 Boolan

Boolan
高端IT咨询与教育平台

现代C++系统研发骨干特训营

模块三、C++内存管理

C++内存管理、策略与优化

智能指针

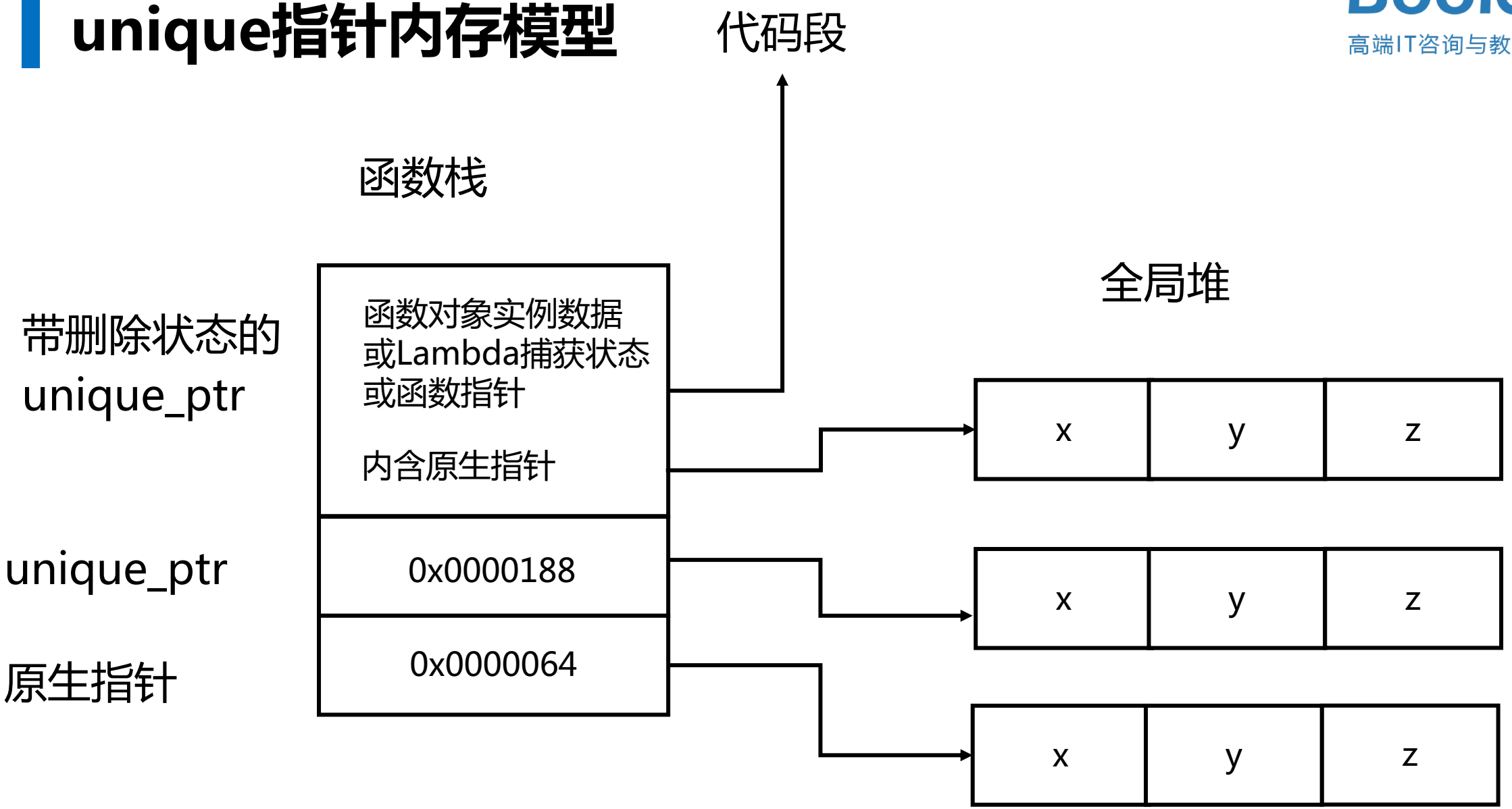
智能指针

- 智能指针封装了裸指针，内部还是裸指针的调用
- 智能指针使用RAII特点，将对象生命周期使用栈来管理。
- 智能指针区分了所有权，因此使用责任更为清晰。
- 智能指针大量使用操作符重载和函数内联特点，调用成本和裸指针无差别

unique_ptr解析

- 默认情况存储成本和裸指针相同，无添加
- 独占拥有权
- 不支持拷贝构造，只支持移动（所有权转移）
- 可以转换成shared_ptr
- 可自定义删除操作（policy设计），注意不同删除操作的存储成本：
  - 函数对象（实例成员决定大小）
  - lambda（注意捕获效应会导致lambda对象变大）
  - 函数指针（增加一个指针长度）

unique指针内存模型



unique_ptr使用API

- uptr.get()　获取原生指针（不能delete），所有权仍归uptr
- uptr.release() 释放所有权、并返回原生指针（要负责delete）
- uptr.reset()，uptr=nullptr 等价，delete堆对象，同时置空指针
- uptr.reset(p) 先delete uptr，再将所有权指向p指针。

- uptr.swap(uptr2) 交换两个智能指针

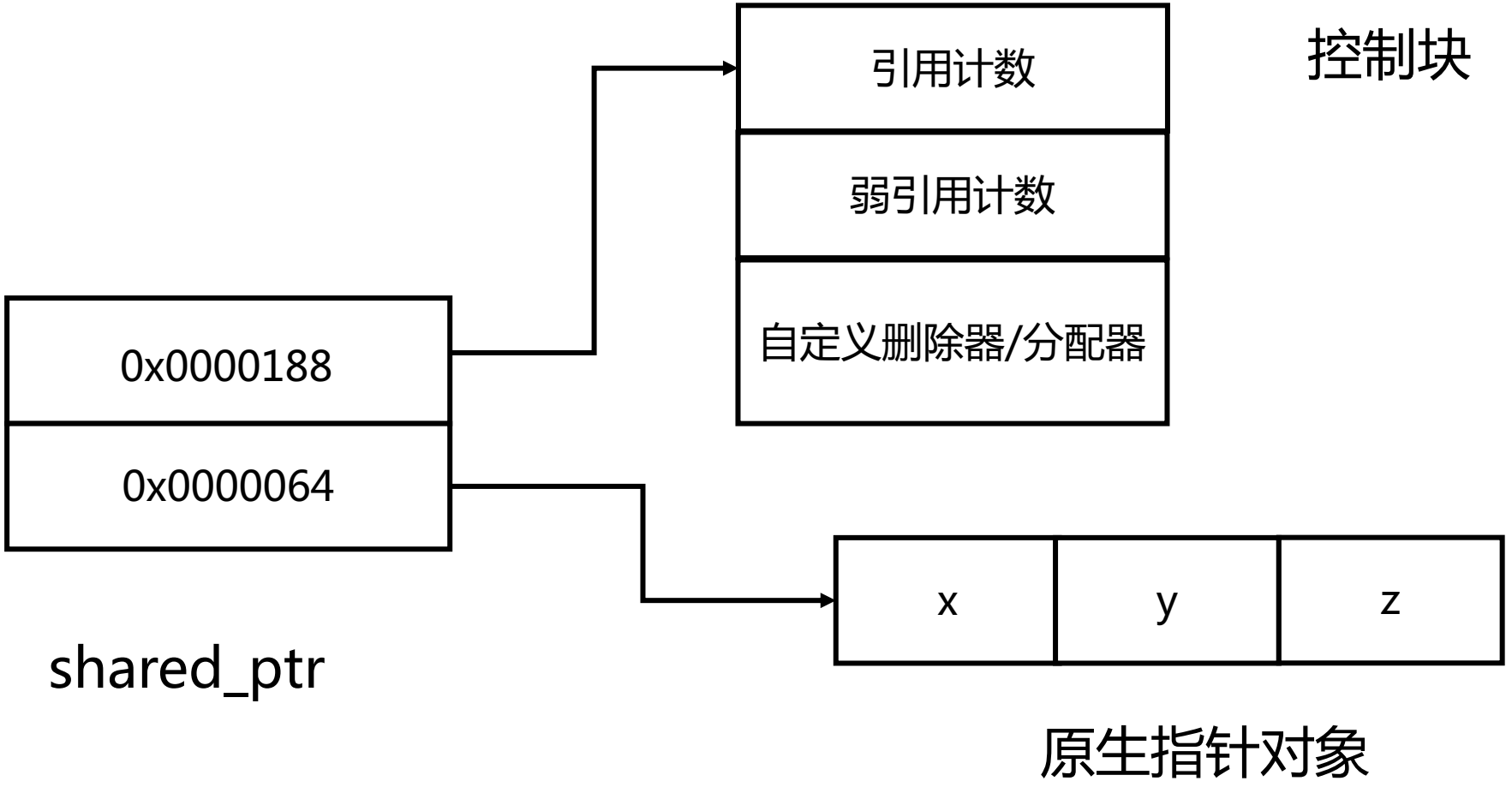
unique_ptr使用场景

- 为动态分配内存提供异常安全（RAII）
- 将动态分配内存的所有权传递给函数
- 从函数内返回动态分配的内存（工厂函数）
- 在容器中保存指针
- 在对象中保存多态子对象（数据成员）

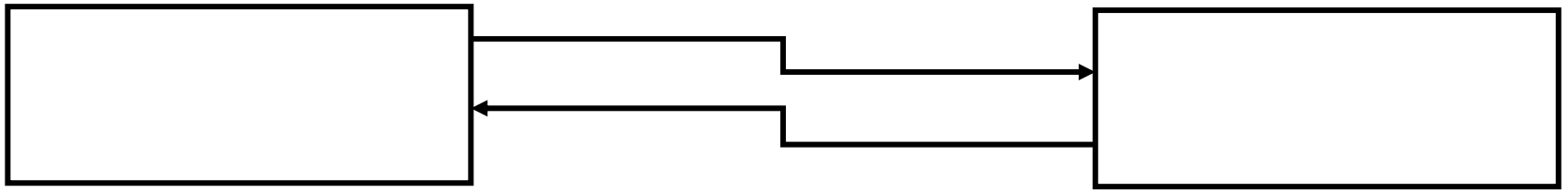
shared_ptr解析

- 共享所有权
- 存储成本较裸指针多了引用计数指针（和相关控制块-共享）
- 接口慎用（蔓延问题）
- 线程安全，引用计数增减会减慢多核性能
- 最适合共享的不变数据
- 支持拷贝构造，支持移动

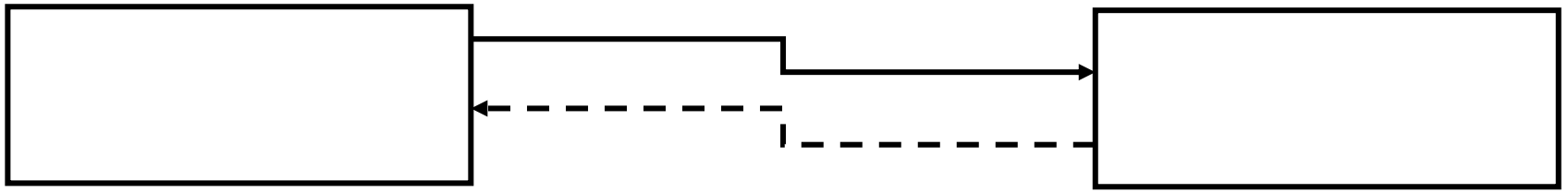
共享指针内存模型



弱引用解除循环引用



共享指针形成循环引用，引用计数都不为0，造成内存泄露



使用弱引用指针，不计引用计数，破解循环引用

其他特性

- 指针的比较操作
- 自定义删除器——注意存储成本增长
- 数组支持（包括operator[] 的支持）
  - unique_ptr<T[]>（C++11）和shared_ptr<T[]>（C++17）

- 避免循环引用
- 避免对原始指针多于两组的所有权
  - 使用enable_shared_from_this<>避免this被多组所有权。

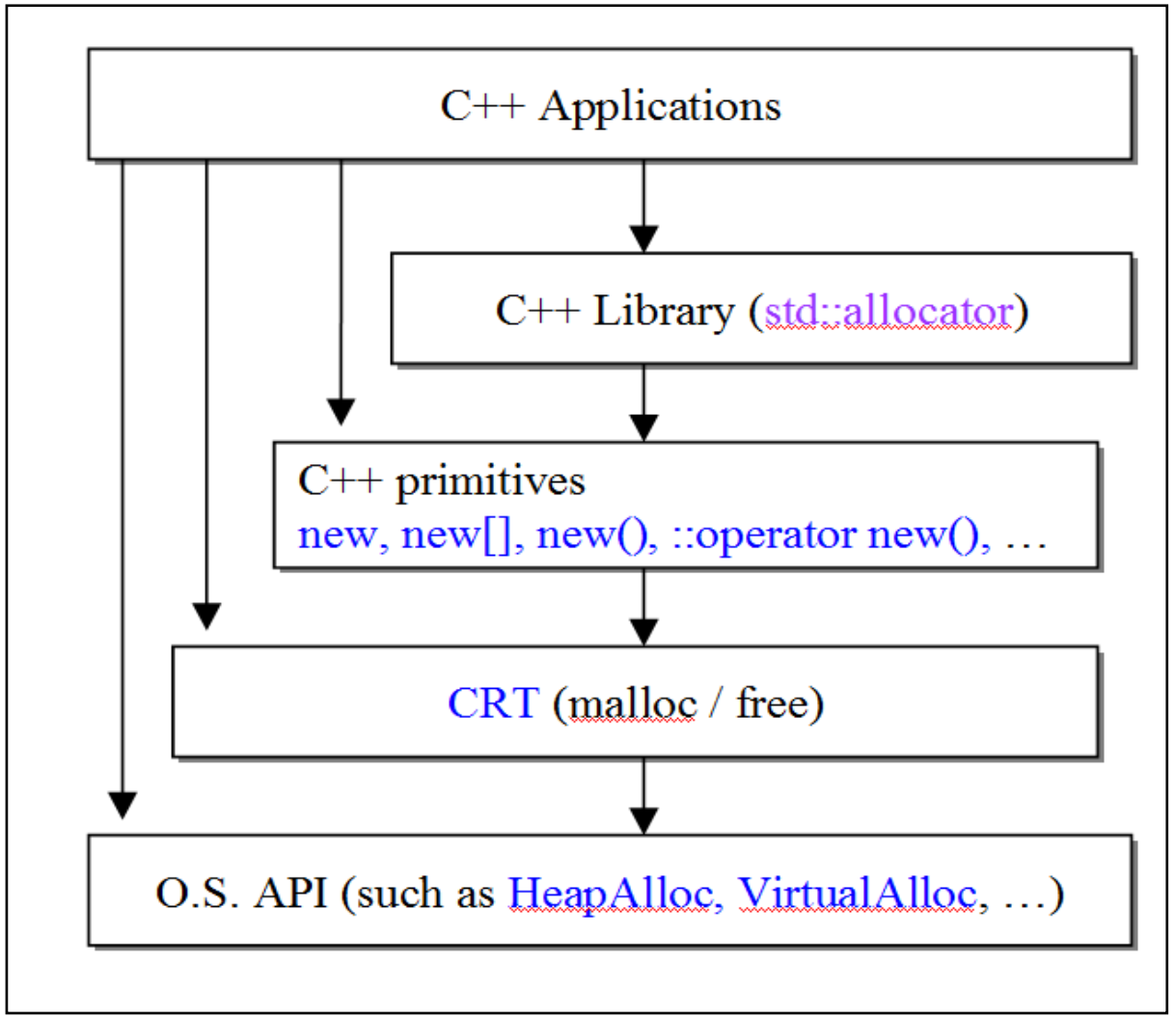
智能指针最佳实践

- 智能指针仅用于管理内存，不要用于管理非内存资源。非内存资源使用RAII类封装
- 用 unique_ptr表达唯一所有权
- 用 shared_ptr表达共享所有权
- 优先采用 unique_ptr 而不是 shared_ptr，除非需要共享所有权
- 针对共享情况考虑使用引用计数。
- 使用 make_unique() 创建 unique_ptr
- 使用 make_shared() 创建 shared_ptr
- 使用 weak_ptr 防止 shared_ptr 的循环引用

智能指针最佳实践

- 原始指针（T*）或引用（T&）没有所有权，
- 以智能指针为参数，仅用于明确表达生存期语义
  - 使用 unique_ptr参数表达所有权转移
  - 使用 shared_ptr参数表达不同共享所有权意图
- 如果不传递所有权，应当接受 T* 或 T& 参数而不是智能指针
- 不要把来自智能指针别名的指针或引用传递出去
- 单个表达式仅进行一次显式资源分配

C++内存分配/释放概览



new/delete 表达式


```
MyClass* myObject = new MyClass{"Software"};
myObject->process();
delete myObject;
```

new 表达式完成两件事情：

- 1. 分配对象所需要的内存
- 2. 调用构造器构造对象初始状态

delete 表达式完成两件事情

- 1. 调用析构器析构对象状态
- 2. 释放对象所占的内存

new[] /delete[] 表达式


```
MyClass* myObject = new MyClass[5];
delete[] myObject;
```

new[] 表达式完成两件事情：

- 1. 分配对象数组所需要的内存
- 2. 每一个元素调用构造器构造对象初始状态

delete[] 表达式完成两件事情

- 1. 每一个元素调用析构器析构对象状态
- 2. 释放对象数组所占的所有内存

new[]/delete[] 不能和new/delete 混搭，必须匹配

placement new

在指定内存位置构造对象

```
void* memory = std::malloc(sizeof(MyClass));
MyClass* myObject = ::new (memory) MyClass("Software");
```

- 只负责构造对象，不负责分配内存
- 没有placement delete，直接显式调用析构器即可。

```
myObject->~MyClass();
std::free(memory);
```

使用STL库函数替换placement new


```
void* memory = std::malloc(sizeof(MyClass));
MyClass* myObject = reinterpret_cast<MyClass*>(memory);

std::uninitialized_fill_n(myObject, 1, MyClass{"Software"});
std::destroy_at(myObject);
std::free(memory);
```

new/delete 操作符

- operator new负责分配内存（当new表达式被调用时）
- 可以定义全局也可以定义针对某一个类的“成对重载”

```
auto operator new(size_t size) -> void* {
   void* p = std::malloc(size);
   std::cout << "allocated " << size << " byte(s)\n";
   return p;
}
auto operator delete(void* p) noexcept -> void {
   std::cout << "deleted memory\n";
   return std::free(p);
}
```

new[]/delete[] 操作符

- new/delete 操作符对应的数组形式, 可以成对重载

```
auto operator new[](size_t size) -> void* {
     void* p = std::malloc(size);
     std::cout << "allocated " << size << " byte(s) new[]\n";
     return p;
}

auto operator delete[](void* p) noexcept -> void {
     std::cout << "deleted memory delete[]\n";
     return std::free(p);
}
```

为某一个类重载new/delete操作符


```
class MyClass {
public:
   auto operator new(size_t size) -> void* {
      return ::operator new(size);
   }
   auto operator delete(void* p) -> void {
      ::operator delete(p);
   }
};
```

重载的情况下，仍然可以随时使用全局的操作符

```
MyClass* p = ::new MyClass{};
::delete p;
```

小对象优化

- 堆分配可能有严重的碎片效应
- 不是所有的new都必然存储在堆上，可以自定义
- 栈适合存储连续的少量对象
- 堆适合存储离散的大量对象
- 利用栈作为对象缓冲区

定制内存管理的非性能理由

即使性能不是一个问题，定制也是有用的。

- 检测内存泄漏。
- 检测多次释放。
- 检测漏写/重写。

定制内存管理的性能理由

- 厂商默认提供的new/new[] 和delete/delete[] 是通用的。
- 必须处理方方面面的情况：
  - 程序运行的时间。
  - 内存分配请求的大小。
  - 动态分配对象的寿命。
  - 线程安全要求。
- 大多数应用程序不需要这样的通用性。

什么时候会因为性能定制内存管理

当出现以下情况，考虑自定义堆管理以提高性能：

- 默认的堆管理是一个瓶颈。
- 可以开发性能更好的实现：
  - 通常是通过消除通用性。
  - 例如，一个线程不安全的分配器，正好用于64字节的请求。

典型定制内存管理提升性能的场景

- 许多对象的生命期同时结束。
  - 所有的对象都可以放在一个堆里（“ arena “），在一次操作中被释放。避免了对单个对象的释放成本。
- 对象自然是一起使用的。
  - 所有对象都可以放在一个堆里（"集群"）。页面故障和缓存缺失减少。
- 代码是单线程的，但默认分配器是线程安全的。
  - ST程序或MT程序中的特定线程分配器。
- 在MT软件中，分配器的争用率很高。
  - 尝试可扩展的分配器，如Hoard、tmalloc、TCMalloc等。
- 几种分配尺寸占主流。
  - 特定大小的分配器（"池"）可以消除大部分碎片。堆管理器的开销（时间+大小）减少了。
- 默认的分配器提供了次优的对齐方式。
  - 在i86上，当8字节对齐时，对double的访问是最快的，但一些默认分配器可能会对它们进行4字节对齐。