使用动态分配变量的C++功能，如标准库容器，智能指针和字符串，使用C++编写应用程序富有成效。但这种表达能力也有黑暗的一面。当性能很重要时，新的就不是你的朋友了

从一个循环或频繁调用的函数中删除一个对内存管理器的调用就足以显著提高性能，而且通常有机会删除多个调用。

线程局域存储期

根据操作系统和编译器的不同，访问线程局部变量的开销可能比访问静态变量的开销更大。

自动存储期

段文字讨论了 C++ 中自动存储期限的概念，它描述了自动变量（通常称为局部变量）的生命周期和它们在内存中的行为。自动变量在函数调用时在栈上分配内存，并在函数返回时释放内存。它们只在声明它们的代码块的作用域内有效。自动变量的内存地址在运行时根据栈指针的当前位置确定，这意味着每次函数调用时自动变量的地址都可能不同。

动态存储

 这段文字描述了 C++ 中动态存储期限的概念，它涉及在堆上分配的内存，这种内存的管理完全由程序控制。动态变量是通过 new 操作符在运行时分配的，并且可以通过 delete 操作符显式地释放。动态变量的生命周期不受作用域限制，它们的存在时间完全取决于程序何时决定释放它们。

值对象和实体对象

文本

描述已自动生成

* 初始化（调用构造函数）
* 赋值（调用赋值运算符）
* 函数参数（每个参数表达式都会被移动或复制构造到它的形式参数中）
* 函数返回（调用移动或复制构造函数，可能两次）
* 将项目插入标准库容器（项目被移动或复制构造）
* 将项目插入向量（如果向量重新分配，所有项目都会被移动或复制构造）

以下是写时复制的步骤：

1. **浅拷贝**：在复制对象时，只是复制了指向数据的指针，而不是数据本身。
2. **延迟深拷贝**：只有在实际需要修改数据时，才会创建数据的深拷贝。
3. **检查引用计数**：在修改操作之前，检查共享指针的引用计数。
4. **唯一化**：如果引用计数大于 1，表示数据被多个对象共享，此时执行深拷贝，并将共享指针更新为指向新的独立副本。
5. **修改数据**：在确认数据不再被共享后，可以安全地修改数据。**对动态分配变量的简单使用是 C++ 程序中最大的性能杀手。当性能至关重要时，new 不是你的朋友。**
6. 动态内存分配（使用 new 和 delete）通常比栈分配（自动变量）要慢，因为它涉及到更复杂的内存管理操作。频繁的动态分配和释放可能会导致性能瓶颈。
7. **开发者可以通过减少调用内存管理器的次数来成为一名有效的优化者。**
8. 减少动态内存分配的次数可以显著提高程序性能。这可以通过重用已分配的内存、使用对象池、或者在栈上分配内存来实现。
9. **程序可以通过提供 ::operator new() 和 ::operator delete() 的定义来全局改变内存的分配方式。**
10. 在 C++ 中，你可以重载全局的 ::operator new() 和 ::operator delete() 来自定义内存分配策略，比如使用定制的内存池。
11. **程序可以通过替换 malloc() 和 free() 来全局改变内存的管理方式。**
12. 在 C 中，可以通过定义 malloc 和 free 的替代函数来改变内存分配行为。在 C++ 中，这通常不推荐，但仍然可行。
13. **智能指针自动化动态变量的所有权。**
14. 智能指针（如 std::unique\_ptr, std::shared\_ptr）可以自动管理动态分配的内存，从而减少内存泄漏的风险。
15. **动态变量的共享所有权更昂贵。**
16. 使用 std::shared\_ptr 管理的共享所有权需要额外的内存来维护引用计数，这可能会增加开销。
17. **静态创建类实例。**
18. 静态分配的对象（例如全局或静态局部变量）只在程序启动时分配一次，可以减少动态分配的开销。
19. **静态创建类成员，如果需要，使用两阶段初始化。**
20. 静态成员可以在类定义之外初始化，并且可以使用两阶段初始化来延迟复杂对象的构造。
21. **使用主指针拥有动态变量，而不是共享所有权。**
22. 使用单一所有权的指针（如 std::unique\_ptr）可以避免引用计数开销，并简化内存管理。
23. **创建无复制函数，通过输出参数传递数据。**
24. 通过返回值或输出参数传递数据可以避免不必要的对象复制。
25. **实现移动语义。**
26. 移动语义允许资源从一个对象转移到另一个对象，而不是复制，这可以显著提高性能。
27. **偏好扁平数据结构。**
28. 扁平的数据结构（例如数组而不是链表）通常有更好的缓存局部性，可以减少内存访问时间。