一、强制性规则

1.命名规则：

类名使用 PascalCase

函数名使用 camelCase

变量名使用小写字母，并且使用下划线分隔单词

常量名使用全大写字母，单词之间用下划线分隔

2.函数和变量的作用域：

函数和变量必须尽可能保持局部性，不使用全局变量（除非在特定的情况下，如日志系统或配置管理）

3.初始化规则：

必须使用统一的初始化方式：使用 uniform initialization（即花括号 {}）初始化所有变量。

4.指针和引用的使用：

对于对象的引用必须使用引用（&amp;）而不是指针（\*），除非有特殊原因

使用智能指针（std::unique\_ptr、std::shared\_ptr）来管理动态内存，避免裸指针

5.异常处理：

必须使用 try-catch 块捕获和处理异常，避免程序异常终止

不允许使用异常来控制程序流程

6.头文件保护：

所有头文件必须使用宏保护，避免重复包含（#ifndef / #define / #endif）

7.头文件依赖管理：

在头文件中避免直接包含 .cpp 文件，减少不必要的依赖

采用前向声明减少头文件包含，特别是对类和结构体

8.内存管理：

使用 RAII (Resource Acquisition Is Initialization) 原则管理资源，不使用裸指针

9.不要使用 using namespace std;：

在全局作用域或头文件中禁止使用 using namespace std;，避免命名冲突

10.代码行长度限制：

每行代码的长度不得超过 120 个字符

11.避免魔法数字：

所有常量值应定义为常量或枚举，避免代码中直接使用数字（例如，10 应该被定义为 MAX\_VALUE）

12.条件表达式：

避免复杂的嵌套条件表达式，分解成简洁易懂的单独语句

13.禁止使用 goto 语句：

不允许使用 goto，避免出现不可预测的程序流

二、推荐性规则

1.命名空间使用：

为了避免命名冲突，建议在项目中使用命名空间（namespace）对不同模块进行划分

2.常量声明：

对于常量，建议使用 constexpr 或 const，而不是 #define

3.枚举类型：

枚举类型应使用 enum class，而不是传统的 enum，以避免隐式转换和作用域污染

4.资源管理：

优先使用 C++11/14 的标准库类型（如 std::vector、std::string、std::unique\_ptr），避免手动管理内存

5.迭代器使用：

使用标准库的迭代器来访问容器元素，而不是索引访问，确保代码的安全性和通用性。

6.多态性：

使用虚函数（virtual）和纯虚函数（= 0）来实现多态，并确保析构函数是虚拟的，以防止对象销毁时出现内存泄漏

7.避免不必要的类型转换：

使用 static\_cast、dynamic\_cast 等明确的类型转换方式，避免隐式类型转换，提升代码的可读性

8.使用 auto 关键字：

尽可能使用 auto 来推导变量类型，减少类型声明的冗余，提高代码的简洁性

9.函数注释：

所有公开函数应具有简洁明了的注释，解释函数的功能、参数和返回值

10.尽量避免过深的嵌套：

在可能的情况下，避免深层嵌套的 if 语句或循环结构，以提高代码的可读性

11.类设计：

应遵循单一责任原则（SRP），每个类应该有清晰的功能，避免类的职责过多

12.模板使用：

避免过度使用模板特化和 SFINAE（Substitution Failure Is Not An Error），当需要时应尽量简化模板代码

13.STL容器的使用：

对于动态数据的存储，优先选择 std::vector 或 std::list，避免使用原生数组

14.头文件的组织：

将接口定义放在头文件中，成员函数的实现放在源文件中

15.避免过多的 inline 函数：

避免在头文件中定义过多的 inline 函数，减少编译器优化开销

三、允许性规则

1.使用 printf 和 scanf：

在调试时，可以使用 printf 和 scanf，但在生产环境中应使用 std::cout 和 std::cin

2.宏定义：

可以使用宏定义，但要尽量减少宏的使用，特别是避免复杂的宏表达式

3.虚拟继承：

虚拟继承可以解决多重继承问题，但应避免复杂的虚拟继承结构，保持继承关系简单

4.静态变量：

静态变量可以用于跨函数和跨文件的数据共享，但应小心使用，以避免引入隐藏的依赖关系

5.编译器扩展：

可以使用编译器的特定扩展来优化性能，但在跨平台的项目中应谨慎使用

6.直接访问私有成员：

可以在测试类中直接访问被测试类的私有成员，虽然这样做不推荐，但在单元测试时可以简化测试过程

7.多线程：

在适当的情况下，可以使用多线程来优化性能，但需考虑线程安全和同步机制

8.继承和多态：

如果需要，允许使用继承和多态，但应避免滥用，导致不必要的复杂性

9.递归：

在某些情况下可以使用递归来简化问题的表达，但需要考虑递归深度和性能影响

10.使用 C 风格字符串：

可以使用 C 风格字符串（char\*），但优先考虑使用 std::string，以避免潜在的内存管理问题