1. 引言

本规范旨在为本团队的 C++ 项目开发提供统一的技术指导，以确保代码质量、可维护性、可读性、性能和安全性。本规范参考了 C++ Core Guidelines。

2. 规范级别定义

a. 强制 (Mandatory): 必须严格遵守的规则。违反这些规则将导致代码审查不通过，甚至可能导致编译错误或运行时问题。

b. 推荐 (Recommended): 强烈建议遵守的规则。遵守这些规则有助于提高代码质量、可读性和维护性。在有充分理由的情况下可以破例，但需在代码审查中说明理由。

c. 允许 (Permitted): 可以在特定情况下选择使用的实践。这些实践通常有其适用场景和局限性，团队成员可根据具体情况自行判断。

3. 规范内容

3.1 通用设计原则

a. 强制

1. 资源管理: 任何资源（内存、文件句柄、网络连接等）的获取都必须与释放配对。优先使用 RAII (Resource Acquisition Is Initialization) 技术，例如 `std::unique\_ptr`、`std::shared\_ptr` 和其他智能指针，以及文件流对象等，确保资源在离开作用域时自动释放。 (C.20, C.21, C.22)

2. 避免裸露的 `new` 和 `delete`: 除了在实现自定义内存管理或底层数据结构时，严禁直接使用 `new` 和 `delete`。 (R.1, R.3)

3. 遵循零开销原则: 语言特性和库的使用不应引入不必要的运行时开销。只为需要的功能付费。 (P.1)

4. 明确所有权: 通过类型系统明确对象的生命周期和所有权。避免所有权模糊不清的代码。 (R.20)

5. 避免空指针解引用: 在解引用指针前，必须进行空值检查或通过设计确保指针始终有效。

6. 错误处理: 必须使用异常处理运行时错误，而不是返回错误码来指示本应阻止程序继续执行的错误。 (E.2)

7. 使用 `const` 关键字: 尽可能地使用 `const` 来声明常量、常量引用和常量成员函数，以增强类型安全和代码可读性。 (Con.1, Con.2)

8. 禁止全局非 `const` 变量: 严禁使用全局非 `const` 变量，以避免潜在的竞态条件和难以调试的问题。 (I.1)

9. 避免宏定义: 尽量避免使用宏，尤其是带有参数的宏。优先使用 `const` 变量、`enum class`、`inline` 函数或 `template` 来替代宏。 (P.10)

10. 头文件自包含: 每个头文件都必须包含它所需要的所有头文件，并且能够独立编译。 (SF.1)

b. 推荐

11. 函数单一职责: 每个函数或方法应该只负责一个明确的职责。 (F.1)

12. 接口最小化: 类和模块应暴露最小化的公共接口，隐藏实现细节。 (C.8)

13. 使用 `override` 和 `final`: 在派生类中重写虚函数时，推荐使用 `override` 关键字。在不希望被进一步派生或重写的类或虚函数上，推荐使用 `final` 关键字。 (C.128, C.129)

14. 避免过长的函数: 推荐将复杂函数拆分为多个小函数，每个小函数完成一个具体的任务。

15. 使用命名空间: 将代码组织在有意义的命名空间中，以避免名称冲突。 (I.8)

16. 使用 `enum class` 而不是 `enum`: 推荐使用强类型枚举 `enum class` 来避免名称污染和隐式转换问题。 (Enum.1)

17. 避免 `using namespace` 在头文件中: 在头文件中，除了在特定的实现文件中明确需要，否则不推荐使用 `using namespace` 声明。 (I.9)

18. 尽可能地使用 `auto` 关键字: 推荐使用 `auto` 关键字来声明局部变量，以提高代码可读性和减少冗余。 (T.1)

19. 避免裸露的 `union`: 除非有充分理由（例如，与 C ABI 兼容），否则不推荐使用裸露的 `union`，这可能导致未定义行为。 (C.183)

20. 优先使用标准库: 优先使用 C++ 标准库提供的容器、算法和工具，而不是自己实现。 (STL.1)

c. 允许

21. 条件性编译 (Conditional Compilation): 允许使用 `ifdef` / `endif` 等宏进行条件编译，但应仅限于平台相关代码或调试代码。

22. 全局 `const` 变量: 允许使用全局的 `const` 变量，但应确保其初始化方式安全且不会引起链接问题。

23. 友元函数 (Friend Functions): 允许在少数特殊情况下使用友元函数，例如当两个类之间有紧密的协作关系，并且友元可以显著简化设计时。 (C.11)

3.2 内存管理与性能

a. 强制

24. 避免内存泄漏: 所有动态分配的内存都必须被正确释放。 (R.7)

25. 区分所有权和非所有权指针/引用: 明确使用 `std::unique\_ptr` 表示独占所有权，`std::shared\_ptr` 表示共享所有权，裸露的指针或引用表示非所有权视图。 (R.20, R.21)

26. 避免裸露的 `malloc`/`free`: 除了与 C 库交互或特定性能优化场景，严禁使用 `malloc`/`free`。 (R.10)

b. 推荐

27. 避免不必要的内存分配: 推荐通过预分配、复用对象或使用栈上的对象来减少堆内存分配的次数。 (Per.2)

28. 按值传递小型且廉价的类型: 推荐按值传递内置类型和小型对象（如 `std::pair`），而不是按 `const` 引用。 (F.16)

29. 按 `const` 引用传递大型或昂贵的类型: 推荐按 `const` 引用传递大型对象或构造昂贵的对象，以避免不必要的拷贝。 (F.17)

30. 使用移动语义: 当处理大型对象时，推荐使用移动语义 (`std::move`, 右值引用) 来避免不必要的拷贝，提升性能。 (R.30)

31. 避免不必要的虚函数: 虚函数引入了动态分派的开销。推荐只在真正需要多态行为时才使用虚函数。 (C.128)

32. 优先使用 `emplace` 而不是 `insert`: 在容器中插入元素时，推荐使用 `emplace` 系列函数，以避免临时对象的构造。 (STL.7)

33. 避免在循环中进行内存分配: 推荐在循环外部进行内存分配，然后在循环内部复用。 (Per.3)

c. 允许

34. 自定义内存分配器: 在有严格性能或内存限制的场景下，允许使用自定义内存分配器，但必须经过严格的代码审查和测试。

35. 指针算术: 在底层数据结构或与硬件交互时，允许使用指针算术，但必须确保其正确性和安全性。

3.3 类与对象

a. 强制

36. 遵守 Big Five/Rule of Zero: 如果一个类需要自定义析构函数、拷贝构造函数、拷贝赋值运算符、移动构造函数或移动赋值运算符中的任何一个，那么它可能需要全部或部分自定义。优先考虑“Rule of Zero”，即如果类没有管理任何资源，则无需自定义任何特殊成员函数。 (C.20)

37. 所有公共继承都是接口继承: 公共继承必须表示“是一个 (is-a)”关系，并且基类应为抽象接口或主要包含虚函数。 (I.2)

38. 禁止在构造函数中抛出异常进行资源管理: 在构造函数中获取资源时，如果发生异常，必须确保已获取的资源能够被正确释放。优先使用 RAII 原则。 (C.30)

b. 推荐

39. 避免公共数据成员: 推荐将类的数据成员声明为 `private` 或 `protected`，并通过公共成员函数访问，以实现封装。 (C.4)

40. 构造函数尽量不执行复杂操作: 推荐构造函数主要用于初始化成员变量，避免执行复杂的业务逻辑或可能失败的操作。

41. 避免循环依赖: 推荐类之间避免形成循环依赖关系，这会导致编译时间增加和设计复杂化。 (I.23)

42. 优先使用组合而不是继承: 推荐优先使用对象组合来复用代码，而不是继承，以降低耦合度。 (I.4)

c. 允许

43. 单例模式: 允许在特定场景下（例如全局配置管理、日志系统）使用单例模式，但应注意其可能带来的测试和并发问题。 (C.150)

44. 友元类 (Friend Classes): 允许在少数特殊情况下使用友元类，例如当两个类高度耦合，并且友元可以显著简化设计时。 (C.11)

3.4. 编码风格与命名

a. 强制

45. 统一代码格式: 必须使用团队约定的代码格式化工具（例如 clang-format）进行代码格式化。

46. 使用有意义的命名: 所有标识符（变量、函数、类、宏等）都必须使用清晰、描述性强的英文命名。

47. 避免缩写: 除非是广泛接受的行业标准缩写，否则应避免过度缩写。

b. 推荐

48. 命名约定:

类名: 使用 `PascalCase` (例如: `MyClass`)。

函数名: 使用 `camelCase` (例如: `myFunction`)。

变量名: 使用 `snake\_case` (例如: `my\_variable`)。

宏定义: 使用全大写和下划线 (例如: `MY\_MACRO`)。

枚举值: 使用全大写 (例如: `ENUM\_VALUE`)。

常量: 使用全大写 (例如: `MAX\_COUNT`)。

49. 注释: 对非自解释的代码段、复杂算法、接口契约和重要设计决策进行注释。注释应保持最新。