2022141461100-欧阳楠-下半期作业

本文档旨在为我们的C++项目制定一套统一的编码标准与最佳实践。遵循这些规范对于确保代码质量、促进团队协作、提升软件的长期可维护性和性能至关重要。

我们的标准植根于清晰性、安全性与效率等核心原则，并大量借鉴了业界公认的最佳实践，例如谷歌C++风格指南和C++核心指南。其目标不仅在于编写功能正确的代码，更在于使其易于理解、调试和演进。

一、目标 C++标准

规则 1 (a. 强制): 所有新代码均应以 C++20 为目标标准。

规则 2 (a. 强制): 除非获得项目领导层的明确批准且编译器支持已普遍可用，否则禁止使用后续标准（如C++23）的特性。

规则 3 (a. 强制): 为确保可移植性和行为的可预测性，严禁使用非标准的编译器扩展 。

二、通用原则

1. 可读性与可维护性至上

规则 4 (b. 推荐): 当面临一个略显“巧妙”或在微观层面性能稍高的

方案与一个更直接、易读的方案之间的选择时，应优先选择可读性，除非存在经过性能分析证实的真正瓶颈。

规则 5 (b. 推荐): 当代码中出现令人意外或不寻常的逻辑时，“为读者留下线索”至关重要，可以通过恰当的注释或通过代码结构本身来揭示其意图。

代码的难懂程度会直接影响开发效率和成本。难以理解的代码在每次开发者与之交互时（无论是调试、修改还是扩展）都会产生一种“认知税”。这种税收表现为时间消耗增加、认知负荷加重以及引入新缺陷的可能性增大。关注可读性和一致性使工程师能够“专注于其他更重要的问题”。因此，投入于编写可读性高的代码，就是对未来生产力和降低维护开销的投资。

2. 一致性

规则 6 (a. 强制): 在整个代码库中保持统一的编码风格，包括格式化、命名、设计模式和错误处理策略等 。

3. 遵循 ISO 标准 C++

规则 7 (a. 强制): 所有代码必须符合指定 ISO C++标准（当前为 C++20）。

规则 8 (a. 强制): 严格禁止使用非标准的编译器扩展或依赖特定编

译器的行为，以确保代码的可移植性并防止供应商锁定 。

4. 清晰直接地表达意图

规则 9 (b. 推荐): 代码的编写方式应使其目的和逻辑尽可能不言自明 。

规则 10 (b. 推荐): 对使用巧妙或不寻常的构造持怀疑和不情愿的态度。尽管 C++允许复杂的习语，但应优先选择其他开发者易于理解的直接解决方案。

三、注释

1. 注释理念：补充而非冗余

规则 11 (b. 推荐): 注释应当阐明代码背后的原因（意图、设计决策）或澄清本质上复杂的逻辑。如果代码本身已经写得很清晰，注释就不应仅仅重述代码正在做什么。重点注释那些不明显的部分、涉及复杂算法、针对微妙问题的变通方法或代表重要设计选择的代码段。

规则 12 (a. 强制): 任何对这些编码标准的已批准偏离（豁免）必须附有注释，解释偏离的理由。

规则 13 (b. 推荐): 将注释集成到编码过程中。在编写或修改代码时撰写注释，而不是事后补充。这能确保注释的准确性并捕捉到即时上下文。

2. 注释风格与格式

规则 14 (b. 推荐): 对单行注释和大多数行尾解释使用 //。

规则 15 (c. 允许): 对多行注释使用 /*... */，例如在开发过程中临时注释掉代码块（尽管版本控制更适合此目的）或用于较大的解释性段落。若使用，请确保风格一致。

规则 16 (b. 推荐): 对于所有公共 API（类、独立函数、公共成员函数），采用$Doxygen$风格的注释（或类似且一致应用的文档生成格式）。这对于维护可理解的接口至关重要。

3. 需要注释的关键区域

规则 17 (b. 推荐): 类声明： 提供类的用途、主要职责以及任何重要的使用说明或其维护的不变式的摘要。

规则 18 (b. 推荐): 函数声明： 对于理解如何正确使用函数而无需阅读其实现至关重要。这包括记录前置条件和后置条件。

规则 19 (b. 推荐): 不明显的变量声明： 如果变量的名称和类型未能完全传达其用途或约束，请添加简短注释。

规则 20 (b. 推荐): 复杂逻辑区段： 对于复杂的算法、状态机或复杂的条件逻辑，提供一个高层次的概述或解释该方法背后的原理。

四、头文件

1. 自包含头文件

规则 21 (a. 强制): 每个头文件必须是自包含的。这意味着它应该能够自行成功编译，而无需在翻译单元中先包含任何其他特定头文件。

规则 22 (a. 强制): 为实现自包含，头文件必须 #include 它直接使用的所有其他提供定义的头文件，或者对仅通过指针或引用引用的类型使用前向声明。

2. 防止多重包含错误

规则 23 (a. 强制): 所有头文件必须使用一种机制来防止在单个翻译单元内多次包含所引发的问题。

3. 前向声明

规则 24 (b. 推荐): 在头文件中，对于仅通过指针或引用使用的类型，优先使用前向声明而非完整的 #include 指令。前向声明可以节省编译时间，因为 #include 会强制编译器打开更多文件并处理更多输入。前向声明可以节省不必要的重新编译。

五、 格式化与风格

1. 行长度

规则 25 (b. 推荐): 遵守最大行长度限制，通常为 100 个字符。过长的代码行需要水平滚动，使代码更难阅读。一致的限制也提高了版本控制和代码审查工具中并排差异视图的可用性。

2. 缩进

规则 26 (a. 强制): 在整个代码库中使用一致的缩进风格。标准是每个缩进级别 2 个空格。

规则 27 (a. 强制): 严格禁止使用制表符（Tabs）进行缩进。仅使用空格。混合使用制表符和空格，或在不同编辑器中使用显示宽度不同的制表符，会导致视觉布局不一致，使代码对不同团队成员难以阅读。空格确保了统一的外观。

3. 大括号放置

规则 28 (a. 强制): if, else, for, while 和 do 语句的主体必须使用大括号，即使主体只包含单个语句。

4. 空格

规则 29 (b. 推荐): 恰当使用空格以提高可读性：在二元运算符周围、参数列表和初始化列表中的逗号之后、for 循环头部的分号之后、基于范围的 for 循环中的冒号周围、if, for, while, switch 等关键字之后。不要在括号与其内容之间，也不要在分号之前使用空格。

5. 函数声明与定

规则 30 (b. 推荐): 将返回类型与函数名放在同一行。

规则 31 (b. 推荐): 对于超出代码行长度限制的长参数列表，后续参数行应进行一致的缩进（通常为 4 个空格或与开括号对齐）。

六、命名约

1. 通用命名理

规则 32 (a. 强制): 必须选择能够清晰指示实体用途或含义的名称。避免过于神秘或过短的名称，除非其作用域非常有限（如循环计数器 i, j）。

规则 33 (a. 强制): 所有名称必须以英文书写，并为保持一致性使用美式英语拼写。

规则 34 (b. 推荐): 避免使用缩写或首字母缩略词，除非它们在项目领域内得到普遍理解或是标准的 C++术语。

2. 文件

规则 35 (b. 推荐): 使用 $lowercase\_with\_underscores$ (例如 file_utility.h, file_utility.cpp)。这是谷歌风格。

3. 类型名称

规则 36 (b. 推荐): 使用 $PascalCase$ (例如 $ErrorCode, MyClass, ConfigurationSettings$)。

4. 全局变量

规则 37 (a. 强制): 尽可能避免使用全局变量。它们破坏封装，使代码更难推理。

规则 38 (c. 允许): 如果不可避免，全局变量必须以 g_ 为前缀 (例如 $g\_applicationInstance$)，以清晰标识其全局作用域。

5. 函数与方法

规则 39 (b. 推荐): 对函数使用 $camelCase$ (例如 $calculateValue()$, $processData()$) 或对行为类似构造函数的函数使用 $PascalCase$。如果是类方法，它们通常遵循变量的命名风格（例如 $camelCase$ 或 $snake\_case$）。

6. 常量名

规则 40 (b. 推荐): 命名常量（包括枚举值）必须全部大写，并使用下划线分隔单词。

7. 命名空间名称

规则 41 (a. 强制): 不要在头文件中使用 using namespace std;（或任何其他宽泛的 *using namespace*）。 讨论了这一点，倾向于使用 *using std*::*cout*; 或完全限定名称。污染全局或宽泛的命名空间可能导致名称冲突 。

规则 42 (b. 推荐): 在 .cpp 文件中，*using namespace* 的使用应最小化并限制在尽可能小的作用域内（例如，函数内部）。 明确警告不要污染全局命名空间，因为在大型代码库中名称冲突的风险很高。命名空间是 C++中防止此类冲突的主要机制。

七、类与结构体

1. 定义用户自定义类型

规则 43 (b. 推荐): 主要对被动数据结构（PDS，类似于 C 风格的结构体）使用 struct，其中所有成员通常都是 public，并且没有复杂的不变式或行为。

规则 44 (b. 推荐): 当需要强制执行不变式、封装数据（即拥有

private 或 protected 成员）或定义管理对象状态和行为的显式构造函数、析构函数或其他成员函数时，使用 class。

2. 成员可见性与封装

规则 45 (a. 强制): 实践强封装。默认情况下将数据成员声明为 private。通过 public 成员函数（访问器/修改器）有选择地公开它们，或者如果打算供派生类使用，则将其声明为 protected，最小化成员的暴露。

3. 构造函数

规则 46 (a. 强制): 确保所有成员变量都由构造函数初始化。优先使用成员初始化列表而非构造函数体内的赋值，以提高效率和正确性。

规则 47 (b. 推荐): 对那些在所有构造函数中都应具有相同默认值的成员使用默认成员初始化器（类内初始化器），以简化构造函数定义。

规则 48 (a. 强制): 如果构造函数无法建立类的不变式并创建一个完全有效的对象，它必须抛出异常。将对象置于部分构造或无效状态是错误的根源。

4. 析构函数

规则 49 (a. 强制): 如果一个类被设计为多态层次结构中的基类（即，它至少有一个 virtual 函数），其析构函数必须声明为 public 和 virtual。这确保了当通过基类指针删除对象时，会调用正确的派生类析构函数，从而防止资源泄漏和未定义行为。

规则 50 (c. 允许): 基类的替代方案（非多态删除）：如果不打算通过基类指针删除基类对象，其析构函数可以是 protected 且非 virtual 的。

5. 继承与多态：override 与 final

规则 51 (a. 强制): 当派生类函数旨在覆盖基类的 virtual 函数时，它必须用 override 说明符标记。这允许编译器验证基类中确实存在具有相同签名的 virtual 函数，从而防止细微的错误。

规则 52 (b. 推荐): 对不打算被后续派生类覆盖的 virtual 函数使用 final 说明符。这清晰地传达了设计意图，并可能启用编译器优化。

规则 53 (a. 强制): 避免在构造函数或析构函数中调用 virtual 函数。在基类构造/析构期间，对象的动态类型是基类的类型，而不是最终派生类的类型，因此 virtual 函数调用将解析为基类的版本。

八、函数与 Lambda 表达式

1. 函数设计原则

规则 54 (b. 推荐): 单一职责： 一个函数应执行单一的、明确定义的逻辑操作。这使得函数更易于理解、测试、重用和维护。

规则 53(b. 推荐): 简洁性： 保持函数简短。庞大、复杂的函数更难阅读，更容易出错，并且通常表明函数试图做太多事情。如果可能，力求函数能在一屏内显示完整。

规则 54(b. 推荐): 参数数量： 保持函数参数数量较少（理想情况下少于四个）。过多的参数可能令人困惑，容易出错，并且可能表明缺少抽象。

2. 参数传递

规则 55 (b. 推荐): “输入”参数 (Input)：对于可廉价复制的类型（如 int, double 等基本类型，指针，引用本身），按值传递。如果函数内部不应修改该值，则按 const 值传递（例如 const int x）。 建议对简单类型使用 const int 而非 const int&。对于其他类型（较大的 struct、class、std::string、std::vector），按 const 引用传递（例如 const std::string& name），以避免昂贵的复制，同时保证原始对象不被修改。

规则 56(b. 推荐): “输入输出”参数 (会被函数修改)：按非 const 引用传递。这向调用者明确表示函数可能会修改该参数。

规则 57 (b. 推荐): 指针 vs. 引用 (可选参数)： 如果 nullptr 是一个有效且有意义的值，表示“无参数”或“可选对象”，则使用指针 (T*)。否则，优先使用引用 (T&)，因为它们不能为空，并且简化了使用。

3. 返回值

规则 58(b. 推荐): 优先按值返回：对于输出值，优先按值返回，而不是使用输出参数。这通常更清晰，并且可以从返回值优化或移动语义中获益以提高效率。

规则 59 (b. 推荐): 返回多个值： 要返回多个值，返回一个带有命名成员的 struct 或 class，或者 std::tuple（或 std::pair）。为清晰起见，通常首选命名的 struct/class。

规则 60 (a. 强制): 绝不返回指向局部对象的指针或引用： 返回指向函数退出时作用域结束的对象的指针或引用，会导致悬空指针/引用和未定义行为。

4. 指定异常保证

规则 61 (b. 推荐): 如果函数保证不抛出异常，或者它在内部处理所有异常，则将其声明为 noexcept。这为调用者提供了重要信息，并

可能启用编译器优化。

规则 62 (b. 推荐): 析构函数、swap 函数、移动构造函数和移动赋值运算符几乎总是应该是 noexcept 的。抛出异常的移动操作会破坏 STL 容器的强异常安全保证，而抛出异常的析构函数可能导致 std::terminate。

九、现代 C++特性

1. auto 类型推导

规则 63 (b. 推荐): 审慎使用 auto 进行类型推导。当显式类型名称冗长且无助于提高清晰度时，或者当类型难以准确拼写时，auto 非常有用。

规则 64 (b. 推荐): 如果一个更具体的命名类型(例如，通过 using 或 typedef 定义的别名）或概念（如果适用）能更清晰地表达变量的意图或约束，则优先使用它们而非 auto。

2. nullptr

规则 65 (a. 强制): 始终使用 nullptr 来表示空指针，而不是 NULL 或整数 0。 nullptr 是类型安全的（它是 std::nullptr_t 类型），可以防止与整数 0 的歧义，从而避免潜在的重载解析问题和错误。

3. 基于范围的 for 循环

规则 66 (a. 强制): 对于遍历整个容器（如 std::vector, std::list, std::map 等）或任何定义了 begin() 和 end() 的序列，优先使用基于范围的 for 循环。它更简洁，更不易出错，并且更清晰地表达了迭代意图。

4. const 与 constexpr

规则 67 (a. 强制):广泛而一致地使用 const。将不应在函数内修改的参数声明为 const。将不修改对象状态的成员函数声明为 const 成员函数。将旨在表示不变值的变量声明为 const。 const 有助于编译器进行优化，更重要的是，它向开发者传达了关于数据和函数行为的重要信息，增强了代码的健壮性和可维护性 。

规则 68 (b. 推荐): 对可以在编译期求值的函数和变量使用 constexpr。constexpr 函数如果使用编译期常量参数调用，其结果可以在编译期计算出来，可用于数组大小、模板参数等。constexpr 变量是编译期常量。这可以提高运行时性能并增强类型安全。

5. 结构化绑定

规则 69 (b. 推荐): 当需要从 std::pair、std::tuple 或具有公共数据成员的简单 struct/class 中提取多个值时，使用结构化绑定。它比单独访问每个成员更简洁、更易读。

6. std::optional, std::variant, std::any

规则 70 (b. 推荐): std::optional<T>：当一个值可能存在也可能不存在时，使用 std::optional<T>。它比使用特殊值来表示缺失更类型安全、更明确。

规则 71 (b. 推荐): std::variant<Types...>：当一个值可以是几种不同类型之一时，使用 std::variant。它避免了传统 union 的许多不安全之处。

规则 72 (c. 允许): std::any：仅在确实需要存储任意类型的值且无法在编译期确定类型时才谨慎使用 std::any。它提供了类型擦除的存储，但代价是类型检查推迟到运行时，且可能涉及动态分配。

7. std::thread, std::mutex, std::atomic, std::async

规则 73 (a. 强制): 当多个线程可能访问共享数据，并且至少有一个线程会修改数据时，必须使用互斥锁 (std::mutex 及其 RAII 包装器 std::lock_guard 或 std::unique_lock) 或原子操作 (std::atomic) 来保护共享数据，以防止数据竞争和未定义行为。

规则 74 (b. 推荐): 仔细设计并发逻辑，以最小化锁的持有时间，避免死锁，并确保线程安全。

十、错误处理

规则 75 (b. 推荐): 使用 assert（或 static_assert 用于编译期检查）来检查编程错误。提及 static_assert 用于编译期检查。断言通常在发布版本中被禁用，因此不应用于处理可预期的运行时错误。

规则 76 (b. 推荐): 对那些即使代码正确也可能发生的运行时错误（例如，文件未找到、网络不可用），使用异常。

十一、测试

1. 单元测试

规则 77 (a. 强制): 所有新的功能性代码必须有配套的单元测试。单元测试验证代码中最小可测试单元（通常是函数或类）的行为是否符合预期。

规则 78 (b. 推荐): 使用标准的 C++单元测试框架。这些框架提供了丰富的功能，如测试发现、断言宏、fixture 管理等，可以简化测试的编写和执行。

2. 集成测试

规则 79 (b. 推荐): 实现集成测试来验证不同组件或模块之间的交互是否正确。集成测试有助于发现单元测试可能遗漏的接口问题或组件协作问题。

3. 测试覆盖率

规则 80 (b. 推荐): 争取较高的测试覆盖率，但更应优先测试关键路径和复杂逻辑。覆盖率是一个有用的指标，但不应是唯一目标；测试的质量同样重要。