# C++ 编码规范要点

一、 头文件管理：清晰与健壮

头文件自包含： 头文件本身应该能独立编译通过。这意味着它需要 #include 所有它直接依赖的其他头文件，不能假设包含它的源文件已经包含了这些依赖。(a)

头文件保护： 必须使用 #define 宏来防止同一个头文件被多次包含。宏的命名格式建议为 \<PROJECT\>\\_\<PATH\>\\_\<FILE\>\\_H\\_（例如 VOS\_INCLUDE\_TIMER\_H）。(a)

包含顺序： 为了提升可读性和减少隐藏依赖，建议按以下顺序包含头文件，各组之间空一行：1) 当前.cpp文件对应的关联头文件；2) C语言标准库头文件（<cstdio>等）；3) C++标准库头文件（<vector>, <string>等）；4) 第三方库头文件；5) 本项目内的其他头文件。(b)

循环依赖： 必须避免头文件之间的循环包含（例如 a.h 包含 b.h，而 b.h 又包含 a.h）。这种依赖会导致编译失败，是设计上的缺陷。(a)

职责单一： 每个头文件应该聚焦于声明一个核心类，或者一组功能紧密相关的函数/类型。避免创建大而全、职责混杂的头文件。(b)

变量定义： 禁止在头文件中定义全局变量（这会导致链接时的多重定义错误）。全局变量应在 .cpp 文件中定义，并在头文件中使用 extern 进行声明。(a)

二、 命名：清晰表达意图

类型命名（类/结构体/枚举/类型别名）： 使用大驼峰式（Pascal Case），例如 ClassName, StructName, EnumType, ValueType。(b)

变量与函数命名：

变量： 使用全小写字母，单词间用下划线连接（snake\_case），如 local\_variable, member\_data。

函数： 使用小驼峰式（camelCase），如 calculateTotal(), getUserName()。

(注：变量和函数的命名规则(b)是分开的，但都强调清晰性)

常量命名： 建议使用 k 前缀加大驼峰式，如 kMaxBufferSize, kDefaultPort。这有助于在代码中清晰区分常量。(b)

类成员变量标识： 需要在命名上区分成员变量和非成员变量。常用方法是在后缀加下划线 (int count\_;) 或加前缀 m\_ (int m\_count)。具体选择哪种方式团队内部统一即可，关键是保持一致性。(b)

宏命名： 全部使用大写字母，单词间用下划线分隔（SCREAMING\_SNAKE\_CASE），例如 #define LOG\_ERROR(msg)。宏因其特殊性，需要格外醒目。(a)

三、 代码格式：一致性与可读性

缩进： 强制使用空格（space）进行缩进，禁止使用制表符（Tab）。每次缩进为 2个空格。(a)

行长度： 推荐每行代码不超过 80个字符。这有助于代码在多种环境和工具（如diff、代码评审界面）中保持可读性。对于长的字符串字面值、#include 路径等特殊情况可以适当放宽。(b)

函数声明与定义：

返回类型通常与函数名放在同一行。

如果参数较多导致一行过长，应将参数分行书写，并进行对齐。例如：

cpp

ReturnType LongClassName::VeryLongFunctionName(Type param1,

Type param2,

Type param3) {

... // 函数体

}

大括号位置：

函数： 左大括号 { 放在函数声明/定义的行尾。如 void foo() { ... }。

类/结构体/命名空间： 左大括号 { 另起新行。

(注意：这里(b)对函数和类的规定略有不同)

条件/循环语句： 强制要求即使循环体或条件分支只有一条语句，也必须使用大括号 {} 包裹。这是为了避免“悬空else”等潜在错误，并提升代码修改的安全性。(a)

指针与引用符号： 推荐将 \* (指针) 和 & (引用) 符号紧贴在类型名后面，而不是靠近变量名。例如 Type\* ptr;, Type& ref;。这种写法更强调类型本身。(b)

四、 注释：有效沟通而非冗余

文件头注释： 推荐在每个源文件（.h, .cpp）开头添加注释块，包含版权信息（如果需要）、作者、文件功能的简要描述、创建或修改日期。可以参考类似华为规范的格式。(b)

函数注释： 对于非平凡的函数，推荐使用注释（特别是支持Doxygen等工具的格式）说明函数的功能、每个参数的含义和约束、返回值的含义、可能抛出的异常或错误条件。例如：

@brief 计算两个数的最大公约数 (GCD).

@param a 第一个正整数.

@param b 第二个正整数.

@return 参数a和b的最大公约数.

@throws std::invalid\_argument 如果a或b不是正整数.

int gcd(int a, int b);

(b)

避免冗余： 严格禁止为显而易见的代码添加注释。例如 i++; // 给i加1 这种注释毫无价值，反而干扰视线。注释应解释“为什么”(Why)和“怎么做”(How)，而不是重复“是什么”(What)。(a)

五、 类设计：封装与清晰

成员顺序： 推荐在类定义中按 public -> protected -> private 的顺序声明成员。在每个访问权限区域内，建议按以下顺序排列：1) 类型定义（typedef, using）和枚举；2) 常量 (static const); 3) 构造函数、析构函数、成员函数（包括静态成员函数）；4) 数据成员（包括静态数据成员）。这样更符合阅读逻辑。(b)

禁用默认拷贝/赋值： 如果一个类不应该被拷贝（例如管理唯一资源的类），必须显式地在 public 区域（或 private 区域，如果使用旧标准）使用 = delete 删除拷贝构造函数和拷贝赋值运算符。依赖编译器自动生成这些函数或只在 private 声明不定义（旧方法）不够明确和安全。(a)

接口类命名： 对于只包含纯虚函数、用于定义抽象接口的类，建议在类名后加上 Interface 后缀，例如 class ReadableInterface。这清晰表明了其角色。(c)

六、 函数设计：简洁与明确

函数长度： 推荐将函数体长度控制在 50行以内。过长的函数通常难以理解和维护。如果函数逻辑复杂，应考虑将其拆分成多个更小、职责更单一的子函数。(b)

const正确性： 强制要求：对于不会修改类成员变量的成员函数，必须声明为 const 成员函数（在函数声明末尾加 const）。这是C++保证对象状态不被意外修改的关键机制。(a)

避免默认参数： 禁止使用函数的默认参数。默认参数会使得函数签名在调用点不够清晰，且可能导致重载解析的歧义。替代方案是使用函数重载（Overloading）来提供不同参数组合的接口。(a)

七、 内存与资源管理：安全与高效

智能指针管理内存： 强制要求：所有通过 new 分配的动态内存资源，其所有权必须立即交给 std::unique\_ptr 或 std::shared\_ptr 进行管理。这是现代C++避免内存泄漏的核心手段。几乎不应该在业务逻辑代码中出现裸的 new/delete。(a)

禁止裸new/delete： 重申：在应用程序层面的代码中，禁止直接使用 new 和 delete 操作原生内存。唯一例外可能出现在底层基础库（如自定义内存池）的实现内部，并且这些操作会被妥善封装隐藏。(a)

八、 错误处理：明确与可控

错误检查： 强制要求：对于返回错误码（或表示错误的状态）的函数，调用方必须立即检查其返回值并进行适当的错误处理。忽略错误返回值是潜在Bug和安全漏洞的温床。(a)

九、 其他关键规则

禁止全局非POD变量： 禁止定义具有非平凡构造/析构（Non-trivial constructor/destructor）的全局或静态类对象（例如 std::string global\_str;）。不同编译单元（.cpp文件）中全局/静态对象的初始化顺序是未定义的（Static Initialization Order Fiasco），极易导致难以调试的运行时问题。如果必须使用全局状态，考虑使用单例模式（需谨慎设计）或在函数内返回局部静态对象（C++11保证线程安全初始化）。(a)

auto使用场景：

允许： 用于明确或冗长的类型，特别是在迭代器场景中非常清晰：auto it = vec.begin(); for (const auto& pair : my\_map)。这提高了代码可读性。

禁止： 禁止在类型信息对理解代码逻辑至关重要时使用 auto，否则会掩盖意图。例如 auto result = Process();，如果 Process() 的返回类型不直观，使用 auto 会让读者难以理解 result 是什么。此时应显式写出类型。(b,a)

范围循环： 推荐在遍历标准容器或数组时使用基于范围的 for 循环 (Range-based for loop)：for (const auto& element : container) { ... }。它更简洁、安全（不易越界），并且意图清晰。(b)

禁用RTTI： 禁止使用运行时类型识别（RTTI）特性，包括 dynamic\_cast 和 typeid。主要原因是它们会带来显著运行时开销，并且通常反映了不佳的设计（过度依赖类型判断而不是多态或模板）。设计上应避免需要运行时类型判断的场景。(a)

避免宏函数： 禁止使用宏（#define）来定义函数样式的代码片段。宏缺乏类型安全、难以调试、容易引入作用域问题。应该使用内联函数（inline）或函数模板（template）来替代。(a)

十、 工具与流程：质量保障

代码格式化工具： 强制要求使用 clang-format 这样的自动化代码格式化工具。团队需要共享并严格遵守同一份格式配置文件（.clang-format）。这能彻底消除代码风格争论，保证代码库风格的高度一致性。(a)

持续集成（CI）检查： 推荐在CI/CD流水线中至少包含以下质量门禁：1) 编译（启用严格警告级别，如 -Wall -Wextra -Werror）；2) 静态代码分析（运行配置好的 clang-tidy/cppcheck）；3) 单元测试执行（确保测试通过）。每次代码提交都应触发这些检查。(b)

代码评审： 强制要求：所有关键代码（涉及核心算法、资源管理、并发控制、安全敏感操作等）在合并到主分支前，必须经过同行代码评审（Code Review）。评审应重点关注逻辑正确性、资源管理（泄漏、所有权）、并发安全性、可读性以及对本规范的遵守情况。(a)