**C++编码规范**

一、文件结构：

1-1.文件命名：

文件名由基本名和后缀名构成，基本名由不大于8个的字母和数字组成，仅允许用小写字母且以字母开头，基本名应是有意义的名字，应与程序功能相一致。后缀名由不超过3个字符组成，常见后缀要求如下：

name.h C ++头文件

name.cpp C++ 源文件

1-2.文件内容：

1-2-1头文件内容：

头文件中一般允许放下列内容：

• 宏定义

• 各种数据结构说明

• typedefs说明

• 外部函数说明

• 全局变量说明

【规范1.1】头文件（\*.h文件）的开始代码部分，一定要加上ifndef/define/ endif等预编译判断条件，防止头文件被重复包含。

【规范1.2】用 #include <filename.h> 格式来引用标准库的头文件（编译器将从标准库目录开始搜索）。

【规范1.3】用 #include “filename.h” 格式来引用自定义/非标准库的头文件（编译器将从用户的工作目录或者指定的路径开始搜索）。

【规范1.4】头文件名不应与标准函数库名相同。

【规范1.5】头文件中只应包括多个文件都需要的内容。对于功能不同的内容应放在不同的头文件中。

1-2-1源程序文件内容：

完整的程序文件由若干部分内容构成，各部分内容及一般顺序如下：

1. 文件头部注释

说明该文件模块的功能和内容（函数、外部数据说明等）。应列文件名、编写日期、作者、主要函数及其功能等。

1. 各种定义及类型定义

Defines和typedefs，其顺序为：

“constant” macro

“function” macro

typedefs

enums

1. 全局（外部）数据说明

Global (external) data declarations , 其顺序为：

externs

non-static globals

static globals

如果一组defines仅应用于某一特定的全局数据块（如标志字），则该defines应紧跟在此数据说明之后，或嵌入到结构说明之中。

1. 函数模块

功能类似的函数应尽量放在一起，每一函数之前应有函数头部注释，主要提供函数的接口说明，内容包括函数基本功能描述、出入口参数、调用关系，必要时也应包括实现算法。

函数体中，根据需要可有代码块注释，它可对某个代码块的功能、编程技巧及临时变量进行说明。

二、排版规则：

【规范2.1】在一个函数体内，逻揖上密切相关的语句之间不加空行，其它地方应加空行分隔。相对独立的程序块之间，变量说明之后必须加空行.

【规范2.2】变量和运算符间要留有空隙，用一个空格符隔开，便于阅读。当然，特殊的单目运算符等除外，如“&”、“.”、“->”、“（）”及作为指针运算符使用时的“\*”。函数定义和调用中若出现多个参数，前一参数后应紧跟逗号运算符，并用一个空格符与后面参数隔开。

【规范2.3】一行代码只做一件事情，如只定义一个变量，或只写一条语句。这样的代码容易阅读，并且方便于写注释。

【规范2.4】if、for、while、do等语句自占一行，除只有一条语句外，执行语句不得紧跟其后，并加{ }。这样可以防止书写失误。

【规范2.5】{ }之内的代码块在“{”右边数格处左对齐。如果出现嵌套的｛ ｝，则使用缩进对齐。

【规范2.6】使用4字符长度的缩进风格，而且最好使用tab键缩进代码。

【规范2.7】关键字之后要留空格。象const、case 等关键字之后至少要留一个空格，否则无法辨析关键字。象if、for、while等关键字之后应留一个空格再跟左括号“（”，以突出关键字。

【规范2.8】函数名之后不要留空格，紧跟左括号“（”，以与关键字区别。

【规范2.9】“赋值操作符、比较操作符、算术操作符、逻辑操作符、位域操作符，如“=”、“+=”、“>=”、“<=”、“+”、“\*”、“%”、“&&”、“||”、“<<”、“^”等二元操作符的前后应当加空格。

【规范2.10】一元操作符如“!”、“~”、“++”、“--”、“&”（地址运算符）等前后不加空格。

【规范2.11】没有必要的话，不要在return语句中使用()。

【规范2.12】左花括号 “{” 放于关键字或方法名的下一行并与之对齐。

【规范2.13】左花括号 “{” 要与相应的右花括号 “}”对齐。

【规范2.14】通常情况下左花括号 “{”单独成行，不与任何语句并列一行。

【规范2.15】尽可能在定义变量的同时初始化该变量（就近原则）。

三、代码注释：

【规范3.1】注释应当准确、易懂，防止注释有二义性，注释率达到20％以上。其中多行注释以/\*……\*/结构，单行以//结构。

【规范3.2】对于主要的宏、常量、结构，在定义时需要加注释。

【规范3.3】全局变量要有较详细的注释，包括对其功能、取值范围、存取时注意事项等的说明。

【规范3.4】重要循环和条件语句一般要写注释，注释放在这段代码之前，同时要保持相同的层次感。典型的算法要编写注释，说明算法的原理、使用的公式、必要时列出参考书籍。

【规范3.5】尽量避免在注释中使用缩写，特别是不常用缩写。

【规范3.6】注释的位置应与被描述的代码相邻，可以放在代码的上方或右方，不可放在下方。

【规范3.7】注释语言用中、英文兼有，建议多使用中文。严禁英文和拼音混用。在使用中文时，必须用 /\* \*/注释，且\*与注释之间留有空格。

【规范3.8】使用良好一致的注释风格，大段注释使用如下格式。除首末行外每行顶头用”\* ”标注。

|  |
| --- |
| 【示例】  /\*  \* This is the correct format for a multiline comment  \*  in a section of code.  \*  \*/ |

【规范3.9】对于一段非常庞大的代码体（比如函数），在其结尾处放置一个识别注释很有好处。

四、命名规则：

【规范4.1】采用大写字母惯例（骆驼式），组成标识符的各个单词以大写字母打头，然后联接起来，同时有一个匈牙利式的前缀。

【规范4.2】程序中不要出现仅靠大小写区分的相似的标识符。

|  |
| --- |
| 【示例】  int x, X; /\* 变量x 与 X 容易混淆 \*/ |

【规范4.3】程序中不要出现标识符完全相同的局部变量和全局变量，尽管两者的作用域不同而不会发生语法错误，但会使人误解。

【规范4.4】尽量避免名字中出现数字编号(标号除外)，如Value1,Value2等，除非逻辑上的确需要编号。

【规范4.5】标识符应当直观且可以拼读，可望文知意。

【规范4.6】标识符最好采用英文单词或其组合，便于记忆和阅读。切忌使用汉语拼音来命名。程序中的英文单词一般不会太复杂，用词应当准确。

【规范4.7】命名长度大致为8~30个字符，使用恰当的缩写。

【规范4.8】关于命名缩写，只有一个限制：如果缩写导致该名字的意义不明确，就不要使用它。

【规范4.9】用全部的大写字母来命名预处理的宏。

【规范4.10】函数应采用动宾结构命名，以动词开头，后跟一个名词，如：Void ResetCounter (Void)。

【规范4.11】变量的名字应当使用“名词”或者“形容词＋名词”。

【规范4.12】局部变量采用短名字。

【规范4.13】不要使用计算机及本专业领域中未被普遍接受的缩写。

【规范4.14】所有单词大写，多个单词之间用 "\_" 隔开。

【规范4.15】类名用大写字母开头的单词组合而成。

五、表达式和基本语句：

【规范5.1】如果代码行中的运算符比较多，用括号确定表达式的操作顺序，避免使用默认的优先级，应当用括号确定表达式的操作顺序。

【规范.2】不要编写太复杂的复合表达式。

【规范5.3】在混合精度的表达式中应该使用同一精度的数据类型进行计算。

【规范5.4】不可将布尔变量直接与TRUE、FALSE 或者1、0 进行比较。

【规范5.5】应当将整型变量用“==”或“！=”直接与0 比较。

【规范5.6】不可将浮点变量用“==”或“！=”与任何数字比较。

【规范5.7】应当将指针变量用“==”或“！=”与NULL 比较。

【规范5.8】不可在for 循环体内修改循环变量，防止for 循环失去控制。

【规范5.9】每个case 语句的结尾不要忘了加break，否则将导致多个分支重叠（除非有意使多个分支重叠）。Switch语句要有CASE配合。

【规范5.10】不要忘记最后那个default 分支。即使程序真的不需要default 处理，也应该保留语句default : break。

【规范5.11】循环、分支层次一般不要超过五层。

六、函数设计：

【规范6.1】函数的规模尽量控制在100行之内（不包括空行和注释）。

【规范6.2】一个函数只完成一项功能。

【规范6.3】参数的书写要完整，不要贪图省事只写参数的类型而省略参数名字。如果函数没有参6，则用void填充。

【规范6.4】如果参数是指针，且仅作输入用，则应在类型前加const，以防止该指针在函数体内被意外修改。

【规范6.5】不要省略返回值的类型。如果没有返回值，应该声明为void类型。

【规范6.6】在函数的“入口处”，需要检查参数的有效性。

【规范6.7】检查通过其它途径进入函数的变量的有效性，如：全局变量、句柄等。

【规范6.8】在C++中，指针的使用应慎重，如果的确只需要借用一下对象的“别名”，提倡使用C++的引用机制，不使用指针。

【规范6.9】一个函数中不能声明另外一个函数。

七、变量、结构

【规范7.1】仔细定义并明确公共变量的含义、作用、取值范围及公共变量间的关系。明确公共变量与操作此公共变量的函数或过程的关系（如访问、修改及创建等）。

【规范7.2】防止局部变量与公共变量同名。防止多个不同模块或函数都可以修改、创建同一公共变量的现象。

【规范7.3】当向公共变量传递数据时，要十分小心，防止赋与不合理的值或越界等现象发生。

【规范7.4】使用严格形式定义的、可移植的数据类型，尽量不要使用与具体硬件或软件环境关系密切的变量。

【规范7.5】结构的设计要尽量考虑向前兼容和以后的版本升级，并为某些未来可能的应用保留余地（如预留一些空间等）。

【规范7.6】合理地设计数据并使用自定义数据类型，避免数据间进行不必要的类型转换。

【规范7.7】常量用全大写字母、下划线分割单词的方式表达。

八、宏

【规范8.1】用宏定义表达式时，要使用完备的括号。

【规范8.2】将宏所定义的多条表达式放在大括号中。

【规范8.3】使用宏时，不允许参数发生变化。

九、类的设计与继承

【规范9.1】弄清楚所要编写的是哪种类。

【规范9.2】用小类代替巨类。

【规范9.3】用组合代替继承。

【规范9.4】避免从并非要设计成基类的类中继承。

【规范9.5】优先提供抽象接口。

【规范9.6】公用继承即可替换性。继承，不是为了重用，而是为了被重用。

【规范9.7】实施安全的改写。

【规范9.8】考虑将虚拟函数声明为非公用的，将公用函数声明为非虚拟的。

【规范9.9】要避免提供隐式转换。

【规范9.10】将数据成员设为私有的，无行为的聚集（C语言形式的struct）除外。

【规范9.11】不要公开内部数据，不要过于自动自发：避免返回类所管理的内部数据的句柄，这样类的客户就不会不受控制地修改对象自己拥有的状态。

【规范9.13】优先编写非成员非友元函数。

【规范9.14】总是一起提供new和delete。

【规范9.15】如果提供类专门的new,应该提供所有标准形式（普通、就地和不抛出）。

十、重载

【规范10.1】运算符既可以定义为全局函数，也可以定义为成员函数，一般应当按照以下规则。

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 【示例】   |  |  | | --- | --- | | 运算符 | 规则 | | 所有的一元运算符 | 建议重载为成员函数 | | = () [] -> | 只能重载为成员函数 | | += -= /= \*= &= |= ~= %= >>= <<= | 建议重载为成员函数 | | 所有其它运算符 | 建议重载为全局函数 | |

【规范10.2】不要改变C++内部数据类型（如int,float等）的运算符。

【规范10.3】不能重载‘.’，因为‘.’在类中对任何成员都有意义，已经成为标准用法。

【规范10.4】不能重载目前C++运算符集合中没有的符号，如#,@,$等。原因有两点，一是难以理解，二是难以确定优先级。

【规范10.5】对已经存在的运算符进行重载时，不能改变优先级规则，否则将引起混乱。

【规范10.6】如果程序同时重载了+=和+，则应该使a += b和a = a + b完成同样的工作。其他类似的操作符重载也一样。

十一、类型安全

【规范11.1】避免使用类型分支，多使用多态。

【规范11.2】依赖类型，而非其表示方式。

【规范11.3】避免对指针使用static\_cast。

【规范11.4】避免强制转换const,强制转换const有时会导致未定义的行为，即使合法，也是不良编程风格的主要表现。

【规范11.5】不要使用C风格的强制转换。

十二、程序可读性

【规范12.1】注意运算符的优先级，并用括号明确表达式的操作顺序，避免是使用默认优先级。

【规范12.2】避免使用不易理解的数字，用有意义的标识来代替。涉及物理状态或含有物理意义的常量，不应直接使用数字，必须使用有意义的枚举或宏来代替。

【建议12.1】源程序中关系较为紧密的代码应尽可能相邻，便于程序阅读和查找。

【建议12.2】不要使用难懂的技巧性很高的语句，除非很有必要时。

十三、程序效率

【规范13.1】编程时要经常注意代码的效率。

【规范13.2】不能为了追求代码的效率，而对软件的正确性、稳定性、可读性造成影响。要在保证软件系统的正确性、稳定性、可读性的前提下，提高代码效率。

【规范13.3】局部效率应为全局效率服务，不能因为提高局部效率而对全局效率造成影响。

【规范13.4】通过对系统数据结构的划分与组织的改进，以及对程序算法的优化来提高空间效率。

【规范13.5】减少循环体内的工作量，提高程序的时间效率。

【建议13.1】仔细分析有关算法，并进行优化。

【建议13.2】对模块中的函数的划分及组织方式进行分析、优化，改进模块中函数的组织结构提高程序效率。

【建议13.3】不应花过多的时间去提高调用不很频繁的函数代码的效率。