**C++编码规范**

**1.编码规范的作用**

对于变成人员，良好的编程风格是提高程序可靠性和效率非常重要的手段。而编码规范就是对编程风格最好的约束保障。   
严格遵守编码规范方便代码的交流和维护，利于提高代码的简洁性，稳定性和效率。

**2.可供参考的C++编码规范**

C++额编码规范设计到程序设计的方方面面，而不是三言两语就可以描述清楚的。下面给出一些具体的编码规范，仅供参考，它说明了编码规范所可能拥有的形式。

**2.1 命名原则**

< 1>减少匿名命名空间级标识符   
< 2>命名时避免使用国际组织占用的格式   
< 3>名字要本着清楚、简单的原则   
< 4>尽量用可发音的名字   
< 5>尽量用英文命名   
< 6>尽量选择通用词汇并贯穿始终   
< 7>避免用模棱两可、晦涩或不标准的缩写   
< 8>避免使用会引起误解的词汇   
< 9>减少名字中的冗余信息   
< 10>建议起名尽量通俗，太专一会限制以后的扩展   
< 11>名字最好尽可能精确地表达其内容   
< 12>避免名字中出现形状混淆的字母或数字   
< 13>命名类和成员使得“object.method()”有意义   
< 14>类和对象名应是名词   
< 15>实现行为的类成员函数名应是动词   
< 16>类的存取和查询成员函数名应是名词或形容词   
< 17>变量名应是名词   
< 18>布尔型的名字要直观   
< 19>避免局部名和外层的名字冲突   
< 20>用a、an、any 区分重名(参数)   
< 21>模板类型名应有意义

此外，经典命名规则主要有：

（1）匈牙利命名法。该命名法是在每个变量名的前面加上若干表示数据类型的字符。基本原则是：变量名=属性+类型+对象描述。如i表示int,所有i开头的变量命都表示int类型。s表示String,所有变量命以s开头的都表示String类型变量。

（2）骆驼命名法。正如它的名称所表示的那样，是指混合使用大小写字母来构成变量和函数的名字。驼峰命名法跟帕斯卡命名法相似，只是首字母为小写，如userName。因为看上去像驼峰，因此而得名。

（3）帕斯卡命名法 即pascal命名法。做法是首字母大写，如UserName,常用在类的变量命名中。

大家可以根据自己的项目类型，选择一个命名规则作为自己的命名方法。

**2.2 函数**

< 1>一定要做到先定义后使用。   
< 2>函数原型申明放在一个头文件中。   
< 3>函数无参数一定要用void标注。   
< 4>对于内置类型参数应传值（除非函数内部要对其修改）。   
< 5>对于非内置类型参数应传递引用（首选）或指针。   
< 6>避免使用参数不确定的函数。   
< 7>若不得不适用参数不确定的函数，用< stdarg.h>提供的方法。   
< 8>避免函数的参数过多。   
< 9>尽量保持函数只有唯一出口。   
< 10>显式定义返回类型   
< 11> (非void)任何情况都要有返回值   
< 12> 若函数返回状态，尝试用枚举作类型   
< 13>返回指针类型的函数应该用NULL 表示失败   
< 14>函数尽量返回引用(而不是值)   
< 15>若必须返回值，不要强行返回引用   
< 16>当函数返回引用或指针时，用文字描述其有效期   
< 17>禁止成员函数返回成员(可读写)的引用或指针   
< 18>重复使用的代码用函数替代

**2.3类的设计申明**

< 1>类应是描述一组对象的集合   
< 2>类成员变量应是私有的(private)   
< 3>保持对象状态信息的持续性   
< 4>提高类内聚合度   
< 5>降低类间的耦合度   
< 6>努力使类的接口少而完备   
< 7>保持类的不同接口在实现原则上的一致性   
< 8>避免为每个类成员提供访问函数   
< 9>不要在类定义时提供成员函数体   
< 10>函数声明(而不是实现)时定义参数的缺省值   
< 11>恰当选择成员函数、全局函数和友元函数   
< 12>防范、杜绝潜在的二义性   
< 13>显式禁止编译器自动生成不需要的函数   
< 14>当遇到错误时对象应该应对有度   
< 15>用嵌套类的方法减少匿名命名空间类的数量

**2.4表达式和控制流程**

< 1>让表达式直观   
< 2>避免在表达式中用赋值语句   
< 3>不能将枚举类型进行运算后再赋给枚举变量   
< 4>避免对浮点类型做等于或不等于判断   
< 5>尝试用范围比较代替精确比较   
< 6>范围用包含下限不包含上限方式表示   
< 7>尽量不适用goto语句   
< 8>在循环过程中不要修改循环计数器

**2.5异常(exception)处理**

< 1>确保代码在异常出现时能正确处理   
< 2>正确注释代码的异常处理能力   
< 3>减少不必要的异常处理   
< 4>不要利用异常处理机制处理其他功能   
< 5>注意模板类型可能会破坏异常处理的一些约定   
< 6>确保异常发生后资源还能被回收   
< 7>特别当心构造函数和析构函数发生异常   
< 8>抛出的异常最好是一个对象   
< 9>捕捉异常时绝不要先基类后派生类   
< 10>捕捉异常时用引用

**2.6头文件**

< 1> 使用条件宏防止头文件多次被引用   
< 2> 确保公共头文件的自足性   
< 3> 只引用需要的头文件   
< 4> 引用头文件的顺序按照googleC++标准，   
参考：[googleC++头文件包含顺序原则](http://blog.csdn.net/K346K346/article/details/47735829)   
< 5> 引用时需要用绝对路径   
< 6> 将函数库放在一个单独的目录下引用   
< 7> 不要在头文件中定义常量/变量   
< 8> 任何声明若被多个源文件引用则应在一个头文件中   
< 9>头文件中尽量避免包含其他的头文件

**2.7性能**

< 1>使用性能追踪分析工具   
< 2>不要用移位代替乘除运算   
< 3>如无必要，不要用非int 的整型类型   
< 4>不要使用关键字register   
< 5>避免在循环体内部定义对象   
< 6>减少代价很高的对象拷贝   
< 7>减少临时对象   
< 8>注意大尺寸对象数组   
< 9>返回对象(值)的优化   
< 10>前缀++和–的效率更高   
< 11> 恰当使用递归   
< 12>恰当地使用inline 函数   
< 13>虚函数和虚继承效率会有一点损失   
< 14>如果合理，使用编译器生成的函数   
< 15>如果合理，构造直传类   
< 16>关于缓存(cache)类成员   
< 17>关于标准库的性能   
< 18>懂得偷懒   
< 19>懂得勤快   
< 20> 80－20原则

**2.8(面向对象的)继承**

< 1>“公共继承(public inheritance)”意味着“派生类是基类”   
< 2>关于“有”和“由…实现”   
< 3>关于继承和模板(template)的区别   
< 4>关于继承接口和继承实现   
< 5>限制继承的层数   
< 6>继承树上非叶子节点的类应是虚基类   
< 7>显式提供继承和访问修饰：public、protected 或private   
< 8>显式指出继承的虚函数   
< 9>基类析构函数(destructor)首选是虚函数   
< 10>绝不要重新定义(继承来的)非虚函数   
< 11>绝不要重新定义缺省参数值   
< 12>不要将基类强制转换成派生类   
< 13>关于C++中的分支用法选择   
< 14>慎用多重继承   
< 15>所有多重继承的基类析构函数都应是虚函数

**2.9内存分配和释放**

< 1>用new、delete 取代malloc、calloc 、realloc 和free   
< 2>new、delete 和new[]、delete[]要成对使用   
< 3>确保所有new 出来的东西适时被delete 掉   
< 4>谁申请谁释放   
< 5>当对象消亡时确保指针成员指向的系统堆内存全部被释放   
< 6>自定义类的new/delete 操作符一定要符合原操作符的行为规范   
< 7>自定义类的new 操作符一定要自定义类的delete 操作符   
< 8>当所指的内存被释放后，指针应有一个合理的值   
< 9>记住给字符串结束符申请空间

**2.10操作符**

< 1> 遵守操作符原本的含义，不要创新   
< 2> 确保自定义操作符能和其他操作符混合使用   
< 3> 区分作为成员函数和作为友元的操作符   
< 4> 尽量使用前缀操作符   
< 5> 确保相关的一组操作符行为统一   
< 6> 绝不要自定义operator&&()、operator||()和operator,()

**2.11类型转换**

< 1> 尽量避免强制类型转换   
< 2> 如果不得不做类型转换，尽量用显式方式   
< 3> 使用新型的类型转换并确保选择正确   
< 4>用虚函数方式取代dynamic\_cast   
< 5>自定义类最好提供显式而不是隐式转换函数   
< 6>用关键字explicit 防止单参数构造函数的类型转换功能   
< 7>限制隐式类型转换的类型数   
< 8>避免多个函数提供相同的类型转换

**2.12编译**

< 1>关注编译时的警告(warning)错误   
< 2>把问题尽量暴露在编译时而不是运行时   
< 3>减少文件的依赖程度   
< 4>减少编译时间   
< 5>透彻研究编译器

**2.13初始化和清除**

< 1> 声明后就初始化强于使用前才初始化   
< 2> 初始化要彻底   
< 3> 确保每一个构造函数都实现完全的初始化   
< 4> 尽量使用初始化列表   
< 5> 初始化列表要按成员声明顺序初始化它们   
< 6> 构造函数没结束，对象就没有构造出来   
< 7> 不要用构造函数初始化静态成员   
< 8> 拷贝构造函数和赋值函数尽量用常量参数   
< 9> 让赋值函数返回当前对象的引用   
< 10>在赋值函数中防范自己赋值自己   
< 11>拷贝和赋值要确保彻底   
< 12>关于构造函数、析构函数、赋值函数、相等或不等函数的格式   
< 13>为大多数类提供缺省和拷贝构造函数、析构函数、赋值函数、相等函数   
< 14>只有在有意义时才提供缺省构造函数   
< 15>包含资源管理的类应自定义拷贝构造函数、赋值函数和析构函数   
< 16>拷贝构造函数、赋值函数和析构函数要么全自定义，要么全生成   
< 17>类应有自己合理的拷贝原则：或浅拷贝或深拷贝   
< 18>若编译时会完全初始化，不要给出数组的尺寸   
< 19>将循环索引的初值定在循环点附近   
< 20>确保全局变量在使用前被初始化

**2.14重载**

< 1> 仔细区分带缺省值参数的函数和重载函数   
< 2> 确保重载函数的所有版本有共同的目的和相似的行为   
< 3> 避免重载在指针和整型类型上   
< 4> 尽量避免重载在模板类型上

**2.15友元**

< 1> 少用友元   
< 2> 减少拥有友元特权的个数

**2.16模板**

< 1> 使用模板如果有限制条件一定要在注释和文档中描述清楚   
< 2> 模板类型应传引用/指针而不是值   
< 3> 注意模板编译的特殊性

**2.17宏**

< 1> 彻底用常量替代(类似功能的)宏   
< 2> 代码中的数值应由一个有意义的标识符代替   
< 3> 若宏值多于一项，一定要使用括号   
< 4> 不要用分号结束宏定义   
< 5> 彻底用inline 函数替代(类似功能的)宏函数   
< 6> 函数宏的每个参数都要括起来   
< 7> 不带参数的宏函数也要定义成函数形式   
< 8> 用{}将函数宏的函数体括起来   
< 9>彻底用typedef 代替宏定义新类型   
< 10>不要在公共头文件中定义宏   
< 11>不要用宏改写语言

**2.18代码格式**

< 1>水平缩进每次用两个空格或一个Tab   
< 2>不要在引用操作符前后加空格   
< 3>不要在单目操作符和其操作对象间加空格   
< 4>不要在“::”前后加空格   
< 5>文件中的主要部分用空行分开   
< 6>函数间要用空行分开   
< 7>用空行将代码按逻辑片断划分   
< 8>花括号{}要单独占一行   
< 9>花括号中没有或只有一条语句时也不省略花括号   
< 10>不要在一行中放多于一条语句   
< 11>语句switch 中的每个case 各占一行   
< 12>语句switch 中的case 按字母顺序排列   
< 13>为所有switch 语句提供default 分支   
< 14>若某个case 不需要break 一定要加注释声明   
< 15>变量定义应集中放置、各占一行，并按字母顺序排列   
< 16>定义指针和引用时\*和&紧跟类型   
< 17>按编译器解析顺序放置变量声明的修饰符   
< 18>函数名和左括号间不要空格   
< 19>声明函数时给出参数的名字，除非没有用处   
< 20>类成员变量和函数按字母顺序排列   
< 21>用带颜色的编辑器

**2.19注释**

< 1> 用英语写全部的注释   
< 2> 确保注释完善你的代码，而不是重复你的代码   
< 3> 注释用词要精确，简单、清楚、切中要害，不能有二义性   
< 4> 注释中的术语要通用   
< 5> 注释中避免引用容易变化的信息   
< 6> 确保所有注释(随代码)及时更新   
< 7> 注释不具备约束使用者行为的能力   
< 8>原则18.10 注释不要嵌套   
< 9>原则18.12 区分“战略性”注释和“战术性”注释   
< 10>行末注释尽量对齐   
< 11>减少不必要的单独占一行的注释   
< 12>对每个#else 或#endif 给出行末注释   
< 13>对每个引用的头文件给出行末注释   
< 14>对每个空循环体给出确认性注释

**3.总结**

要想写出优秀的C/C++代码有很多注意点，远远不是上面这些规则所能完全囊括的。并且，这里罗列的编码规范可能与你曾经见过的一些编码规范所有抵触，这很正常。所以，如果你相信自己的代码是简洁高效稳定的，不必拘泥与这些特定的编码规范。编码规范的灵活是隐藏在编码规范之后所要达到的真正目的。