项目开发标准

V1.0.0

Va.b.c 中 a 为大版本号，b 为项目更迭号，c 作为标识。c：0 为测试版本，1 为稳定版本参考

前言

本规范制定了编写 C、C++、Python 语言程序的基本原则、规则和建议。从代码的清晰、简洁、可测试、安全、程序效率、可移植方面对编程作出具体指导。
考虑到开发者未经历项目开发过程，缺乏项目经验，为提高代码质量，本人编写本规范以指导开发人员编写代码。编程规范 1.0.1 由我本人分析总结典型问题，参考业界编程规范成果，如华为的 C 语言规范，Google 的 C++规范，重新进行了梳理所编写。

清晰第一

清晰性是易于维护、易于重构的程序必需具备的特征。代码首先是给人读，给人看的。
一般情况下，代码的可阅读性高于性能，只有确定性能是瓶颈时，才应该主动优化。

简洁为美

简洁就是易于理解并且易于实现。代码越长越难以看懂，也就越容易在修改时引入错误。写的代码越多，意味出错的地方越多，也就意味着代码的可靠性越低。因此，推荐大家通过编写简洁明了的代码来提升可靠性。

选择合适的风格，与代码原有风格保持一致

所有人共同使用一种风格远远好过为了统一而刻意违背自己的习惯。在已有的规范下，审慎地编些代码以使代码尽可能清晰，是一项非常重要的技能。如果重构/修改其他风格的代码时，比较明智的做法是根据现有代码的现有风格继续编写代码，或者使用格式转换工具进行转换。

规范实施、解释

本规范制定了编写程序的基本原则、规则和建议。
本规范适用于使用 C，C++，Python 语言编码的所有项目。
本规范自发布之日起生效，之后新编写的和修改的代码应遵守本规范。
在某些情况下需要违反本文档给出的规则时，相关项目组自行处理，但是需要提交开发、维护文档，否则实验室将不再为该项目提供维护和服务支撑，个体开发者不得违反本规范中的相关规则或项目组规范。
注意:本规范并非教程,我们假定读者已经对规范内涉及的语言或技术非常熟悉

--陈旭

目录

一．宏规范

（一）语言标准

1. 嵌入式开发

(1) C

① 对于 51 单片机，使用 C51，不兼容包括 C89 及其之后的 C|C++标准，一个标准 C 的微集。

② 对于 stm32，使用 C89，不推荐 C99

③ 对于 ARM，使用 C89，支持 C99，不推荐使用 C++

(2) Python

对于 openMV，使用 microPython

2. 应用开发

（1）C

使用 C99 及其更高的语言标准，

（2）C++

使用 C++11 及其更高的语言标准，如果不得不回滚版本，务必在开发手册注明

（3）Python

使用 Python3 及其更高的语言标准，如果不得不回滚版本，务必在开发手册注明

（二）开发工具链

1.嵌入式开发

① C51 使用 keil-C51

② Stm32 使用 keil-MDK

③ ARM 使用 keil-ARM

④ openMV 使用 openMVIDE

2.应用开发

（1）C

① 编译器使用 gcc，允许 clang，MSVC。

② 对于 IDE，使用 code：blocks，VScode，允许 Embarcadero C++ Builder，Visual Studio，使用 VS 注意 Microsoft 自标准

③ 链接器使用 Cmake

（2）C++

① 编译器使用 g++，允许 clang，MSVC。

② 对于 IDE，使用 code：blocks，VScode，允许 Embarcadero C++ Builder，Visual

Studio，使用 VS 注意 Microsoft 自标准
链接器使用 Cmake
（3）Python
对于 IDE，使用 Spyder，Jupyter Notebook，VScode，允许 Pycharm 等，不推荐 IDLE
（4）图形库
使用 OpenCV，允许 Cimg，不推荐 MATLAB
（5）矩阵库
使用 numpy，numcpp，允许 Eigen3，Armadillo，不推荐 OpenCV 矩阵库
1. 环境约定
（1）人工智能环境

- TensorFlow 框架使用 TensorFlow2（抛弃 TensorFlow1.x）
- Python 3.6-3.9
- Ubuntu 16.04 或更高版本
- Windows 7 或更高版本
- macOS 10.12.6 (Sierra) 或更高版本（不支持 GPU）
- NVIDIA CUDA 版本不低于 11.2
- 驱动程序要求 450.80.02 或更高版本
- cudnn 要求不低于 8.1.0
- AMD GPU 要使用 rocm 库

注：如果使用服务器训练，务必联系系统管理员查看驱动版本，禁止私自迭代 NVIDIA 驱动
（2）包管理及虚拟环境
包管理使用 conda，Ubuntu 允许使用 Python3 自建虚拟环境用 pip 配置。
conda 中 base 环境禁止开发，开发时应当建立虚拟环境，并选择 Python 版本
conda 建议安装 miniconda，允许 anaconda。
（3）图形库
OpenCV 使用 4.5.4，注意 conda 无法安装，可以在 conda 环境下使用 pip 安装

（三）规范语言约定

使用：编程时优先服从
允许：尊重个人习惯，使用后对项目实现影响不大，建议测试人员驳回提交
不推荐：使用会导致出现较为严重后果，原则上测试人员应据此驳回提交
禁止：使用后会造成严重损失和系统威胁，审核人员务必驳回，并提交备案至项目负责人
原则：编程时必须坚持的指导思想
务必：编程时强制必须遵守的约定
建议：编程时必须加以考虑的约定
说明：对此原则/规则/建议进行必要的解释
示例：对此原则/规则/建议从正、反两个方面给出例子

一．开发规范

（一） C

1.头文件规范

a 原则

1 头文件中适合放置接口的声明，不适合放置实现

说明：头文件是模块对外接口。头文件中应放置对外部的声明，如对外提供的函数声明、宏定义、类型定义等。

内部使用的函数（相当于类的私有方法）声明不应放在头文件中。

内部使用的宏、枚举、结构定义不应放入头文件中。

变量定义不应放在头文件中，应放在.c 文件中。

变量的声明不要放在头文件中，即不要使用全局变量作为接口。变量是内部实现细节，不应通过在头文件中声明的方式暴露，应通过函数接口的方式进行对外暴露。即使必须使用全局变量，也只应当在.c 中定义全局变量，在.h 中仅声明变量为全局的。

2 头文件应当职责单一

说明：头文件过于复杂，依赖过于复杂会导致编译时间过长。很多现有代码中头文件过大，职责过多，再加上循环依赖的问题，可能导致为了在.c 中使用一个宏，而包含十几个头文件。模块划分应该在项目自顶向下设计时完成，如果模块为底层独立模块（稳定模块），可以使用.hpp 封装

3 头文件应向稳定的方向包含

b 规则

1 每一个.c 文件应有一个同名.h 文件，用于声明需要对外公开的接口

说明：如果一个.c 文件不需要对外公布任何接口，则.h 就不应当存在，除非是程序的入口，如 main 函数所在的文件。

不推荐一个.c 文件对应两个头文件，一个用于存放对外公开的接口，一个用于存放内部需要用到的定义、声明等，以控制.c 文件的代码行数。源文件过大，应首先考虑拆分.c 文件，一旦把私有定义、声明放到独立的头文件中，就无法从技术上避免别人 include，难以保证定义私有。

2 禁止头文件循环依赖

说明：头文件循环依赖，指 a.h 包含 b.h，b.h 包含 c.h，c.h 包含 a.h 之类导致任何一个头文件修改，都导致所有包含了 a.h/b.h/c.h 的代码全部重新编译一遍。而如果是单向依赖，如a.h 包含 b.h,b.h 包含 c.h,而 c.h 不包含任何头文件,则修改 a.h 不会导致包含了 b.h/c.h 的源代码重新编译。

3 .c/.h 文件禁止包含用不到的头文件

说明：系统头文件包含关系复杂，不能为了省事起见，直接包含一切想到的头文件，然后使用，这种只图一时省事的做法，会导致整个系统的编译时间进一步恶化，并对后人的维护造成了巨大麻烦。

4 头文件应当自包含

说明：简单的说，自包含就是任意一个头文件均可独立编译。

示例：如果 a.h 不是自包含的，需要包含 b.h 才能编译，会带来的危害：每个使用 a.h 头文件的.c 文件，为了让引入的 a.h 的内容编译通过，都要包含额外的头文件 b.h。额外的头文件 b.h 必须在 a.h 之前进行包含，这在包含顺序上产生了依赖。

注意：规则 3 同样适用，不能为了让 a.h 自包含，而在 a.h 中包含不必要的头文件。a.h 要刚刚可以自包含，不能在 a.h 中多包含任何满足自包含之外的其他头文件。

5 总是编写内部#include 保护符（#define 保护）

说明：多次包含一个头文件可以通过设计来避免。如果不能做到，就需要采取阻止头文件内容被包含多于一次的机制。

通常的手段是为每个文件配置一个宏，当头文件第一次被包含时就定义这个宏，并在头文件被再次包含时使用它以排除文件内容。

所有头文件都应当使用#define 防止头文件被多重包含，命名格式为 FILENAME_H，为了保证唯一性，更好的命名是 PROJECTNAME_FILENAME_H。

注：不推荐在宏最前面加上“_”，因为一般以"_"和”__"开头的标识符为系统保留或者标准库使用，在有些静态检查工具中，全局可见的标识符以"_"开头会抛出警告。

注意：不要在受保护部分的前后放置代码或者注释。

示例代码：

```
#ifndef PROJECTNAME_FILENAME_H
#define PROJECTNAME_FILENAME_H
..
#endif
```

6 禁止在头文件中定义变量

说明：在头文件中定义变量，将会由于头文件被其他.c 文件包含而导致变量重复定义。

7 只能通过包含头文件的方式使用其他.c 提供的接口，禁止在.c 中通过 extern 的方式使用外部函数接口、变量

说明:若 a.c 使用了 b.c 定义的 foo()函数,则应当在 b.h 中声明 extern int foo(int input);并在 a.c 中通过#include <b.h>来使用 foo。禁止通过在 a.c 中直接写 extern int foo(int input);来使用 foo，后面这种写法容易在 foo 改变时可能导致声明和定义不一致。

8 禁止在 extern "C"中包含头文件

说明：在 extern "C"中包含头文件，会导致 extern "C"嵌套，Visual Studio 对 extern "C"嵌套层次有限制，嵌套层次太多会编译错误。

c 建议

1 一个模块通常包含多个.c 文件，建议放在同一个目录下，目录名即为模块名。为方便外部使用者，建议每一个模块提供一个.h，文件名为目录名

说明：需要注意的是，这个.h 并不是简单的包含所有内部的.h，它是为了模块使用者的方便，对外整体提供的模块接口。

有些模块，内部功能相对松散，可能并不一定需要提供这个.h，而是直接提供各个子模块或者.c 的头文件。

2 如果一个模块包含多个子模块，则建议每一个子模块提供一个对外的.h，文件名为子模块名

说明：降低接口使用者的编写难度。

3 头文件不要使用非习惯用法的扩展名如：.inc

说明：不推荐.inc 作为头文件扩展名，这不符合 c 语言的习惯用法。不提倡将私有定义单

独放在头文件中
除此之外，Visual stduio 等 IDE 工具无法识别其为头文件，导致很多功能不可用，如“跳转到变量定义处”。虽然可以通过配置，强迫 IDE 识别.inc 为头文件，但是有些软件无法配置，如 Visual Assist

4 统一包含头文件排列方式。

说明：常见的包含头文件排列方式有功能块排序、文件名升序、稳定度排序等。以升序方式排列头文件可以避免头文件被重复包含，以稳定度排序，建议将不稳定的头文件放在前面。

2. 函数规范

a 原则

1 一个函数仅完成一件功能。

说明：一个函数实现多个功能给开发、使用、维护都带来很大的困难。
将没有关联或者关联很弱的语句放到同一函数中，会导致函数职责不明确，难以理解，难以测试和改动。
注：不推荐使用 realloc()函数，因为承担了太多的其他任务

2 重复代码应该尽可能提炼成函数。

说明：重复代码提炼成函数可以带来维护成本的降低。
在“代码能用就不改”的指导原则之下，新需求增加带来的代码“增量迭代”，随着时间的迁移，成品中堆砌着许多类似或重复代码。
开发组应当使用代码重复度检查工具，在持续集成环境中持续检查代码重复度指标变化趋势，对新增重复代码及时重构。当一段代码重复两次时，应考虑消除重复，当代码重复超过三次时，务必立刻着手消除。
一般情况下，可以通过提炼函数的形式消除重复代码。

b 规则

1 避免函数过长，新增函数不超过 50 行

说明：仅对新增函数做要求，对已有函数修改时，建议不增加代码。业界普遍认为一个函数的代码行不要超过一个屏幕，避免来回翻页影响阅读。一般代码度量工具建议都对此进行检查，Logiscope 的函数度量-"Number of Statement" （函数中的可执行语句数）建议不超过 20 行，QAC 建议一个函数中的所有行数（包括注释和空白行）不超过 50 行。
过长的函数往往意味着函数功能不单一，过于复杂，某些实现算法的函数例外，由于算法的聚合性与功能的全面性，可能会超过。

2 避免函数嵌套过深，新增函数代码嵌套不超过 4 层

说明：仅对新增函数做要求，对已有的代码建议不增加嵌套层次。
函数的代码块嵌套深度指的是函数中的代码控制块（例如：if、for、while、switch 等）之间

互相包含的深度。每级嵌套不光会增加代码的晦涩程度，还让程序执行时不能完全利用现代 CPU 流水线分支预测加速。此外避免代码的阅读者一次记住太多的上下文。

3 可重入函数应避免使用共享变量；需要使用应通过互斥手段（关中断、信号量）加以保护

说明：可重入函数是指可能被多个任务并发调用的函数。在多任务操作系统中，函数具有可重入性是多个任务可以共用此函数的必要条件。共享变量指的全局变量和 static 变量。

编写 C 语言的可重入函数时，不应使用 static 局部变量，否则必须经过特殊处理，才能使函数具有可重入性。

4 对参数的合法性检查，由调用者负责还是由接口函数负责，应在项目组/模块内统一规定，缺省由调用者负责

说明：对于模块间接口函数参数的合法性检查，往往有两个极端，要么是调用者和被调用者对参数均不作合法性检查，就遗漏了合法性检查这一必要的处理过程，造成问题隐患；要么就是调用者和被调用者均对参数进行合法性检查，产生冗余代码，降低了效率。

5 对函数的错误返回码要全面处理

说明：一个函数（标准库中的函数/第三方库函数/用户定义的函数）能够提供一些指示错误发生的方法。这可以通过错误标记、特殊的返回数据或其他手段，不管什么时候函数提供了这样的机制，调用程序应该在函数返回时立刻检查错误指示。

6 设计高扇入，合理扇出（小于 7）的函数

说明：扇出是指一个函数直接调用（控制）其它函数的数目，而扇入是指有多少上级函数调用它。

扇出过大，表明函数过分复杂，控制和协调下级函数过多；扇出过小，表明函数的调用层次过多，不利于程序阅读和函数结构分析，并且运行时会对系统资源（如堆栈空间）造成压力。通常函数比较合理的扇出（调度函数除外）通常是 3~5。

注：扇出太大可适当增加中间层次的函数。扇出太小可把下级函数进一步分解或合并到上级函数中。

扇入越大，表明使用此函数的上级函数越多，函数使用效率高，如公共模块函数及底层函数一般有较高扇入。通常顶层函数扇出较高，中层函数扇出较少，而底层函数则扇入到公共模块。

7 废弃代码（没有被调用的函数和变量)要及时清除

说明：废弃代码占用额外空间，还影响程序功能与性能，给程序测试、维护等造成不必要的麻烦。

c 建议

1 函数不变参数使用 const

说明：不变的值更易于跟踪、分析，把 const 作为默认选项，在编译时会对其进行检查，使代码更安全。

2 函数应避免使用全局变量、静态局部变量和 I/O 操作，不可避免的地方应集中使用

说明：带有内部“存储器”的函数的功能可能是不可预测的，因为它的输出可能取决于内部存储器（如某标记）的状态。这样的函数既不易于理解又不利于测试和维护。在 C 语言中，函数的 static 局部变量是函数的内部存储器，有可能使函数的功能不可预测，然而，当某函数的返回值为指针类型时，则必须是 static 的局部变量的地址作为返回值，若为 auto 类，则返回为错针。

3 检查函数所有非参数输入的有效性，如数据文件、公共变量

说明：函数的输入主要有两种：一种是参数输入；另一种是全局变量、数据文件的输入，即非参数输入。函数在使用输入参数之前，应进行有效性检查。

4 函数的参数个数不超过 5 个

说明：函数的参数过多，会使得函数易于受外部（其他部分的代码）变化的影响，从而影响维护工作。函数的参数过多同时也会增大测试的工作量。 函数的参数个数不要超过 5 个，如果超过了建议拆分为不同函数。

5 除打印类函数，不要使用可变长参函数。

说明：可变长参函数的处理过程比较复杂容易引入错误，而且性能也比较低，使用过多的可变长参函数将导致函数的维护难度大大增加。

6 在源文件范围内声明和定义的所有函数，除非外部可见，否则应增加 static 关键字

说明：如果函数只在同一文件中其他地方调用，那么就用 static 声明。使用 static 确保只是在声明它的文件中是可见的，避免其他文件或库中的相同标识符发生混淆。

3. 标识符规范

标识符命名规则历来是一个敏感话题，典型命名风格如 unix 风格、windows 风格，从来无法达成共识。实际上，各种风格都有优势也有劣势，而且也和个人审美有关。我对标识符规范主要是为了让团队的代码看起来统一，有利于代码的后续阅读和修改，项目组可以根据自己的实际需要指定命名风格，规范中不再做统一的

规定。但是注意，项目组内部命名文档要备案至数据中心，或参与下个版本规范迭代。

--陈旭

a 原则

1 标识符的命名要清晰、明了，有明确含义，同时使用完整的单词或大家基本可以理解的缩写，避免使人产生误解。

说明：尽可能给出描述性名称，不要节约空间，让别人很快理解你的代码更重要。变量名只是给人看的，变量名长度不会影响程序运行效率，当然过长的名称会增加编译成本。

2 除了常见的通用缩写以外，不使用单词缩写，不得使用汉语拼音。

说明：较短的单词可通过去掉“元音”形成缩写，较长的单词可取单词的头几个字母形成缩写，一些单词有大家公认的缩写，常用单词的缩写必须统一。协议中的单词的缩写与协议保持一致。对于某个系统使用的专用缩写应该在注视或者某处做统一说明。

示例：一些常见可以缩写的例子：

- argument 可缩写为 arg
- buffer 可缩写为 buff
- clock 可缩写为 clk
- command 可缩写为 cmd
- compare 可缩写为 cmp
- configuration 可缩写为 cfg
- device 可缩写为 dev
- error 可缩写为 err
- hexadecimal 可缩写为 hex
- increment 可缩写为 inc
- initialize 可缩写为 init
- maximum 可缩写为 max
- message 可缩写为 msg
- minimum 可缩写为 min
- parameter 可缩写为 para
- previous 可缩写为 prev
- register 可缩写为 reg
- semaphore 可缩写为 sem
- statistic 可缩写为 stat
- synchronize 可缩写为 sync
- temp 可缩写为 tmp

b 规则

1 项目组内部应保持统一的命名风格。

说明：Unix like 和 windows like 风格均有其拥趸，项目组根据自己的部署平台，以及喜好选择其中一种，并在产品内部保持一致。

2 全局变量应增加“g_”前缀。

3 静态变量应增加“s_”前缀。

说明：全局变量是十分风险的变量，通过前缀可以使全局变量更醒目，促使开发人员更加小心。

4 禁止使用单字节命名变量，但运行定义 i、j、k 作为局部循环变量。

5 对于数值或者字符串等等常量的定义，建议采用全大写字母，单词之间加下划线“_”的方式命名（枚举同样建议使用此方式定义）。

6 除了头文件或编译开关等特殊标识定义，宏定义不能使用下划线“_”开头和结尾

说明：一般来说，”_”开头、结尾的宏都是一些内部的定义

c 建议

1 用正确的反义词组命名具有互斥意义的变量或相反动作的函数等。

示例：add/remove begin/end create/destroy insert/delete first/last
get/release increment/decrement put/get add/delete lock/unlock
open/close min/max old/new start/stop next/previous source/target
show/hide send/receive source/destination copy/paste up/down

2 尽量避免名字中出现数字编号，除非逻辑上的确需要编号

3 标识符前不添加模块、项目的名称作为前缀。

说明：不推荐在文件名中增加模块名，这种写法类似匈牙利命名，导致文件名不可读

有如下问题：

a. 第一眼看到的是模块名，而不是真正的文件功能，阻碍阅读；
b. 文件名太长；
c. 文件名和模块绑定，不利于维护和移植。示例：foo.c 进行重构后，从 a 模块挪到 b 模块，若 foo.c 中有模块名，则需要将文件名从 a_module_foo.c 改为 b_module_foo.c

4 平台/驱动等适配代码的标识符命名风格保持和平台/驱动一致。

说明：涉及外购芯片以及配套驱动，这部分的代码变动（包括为产品做适配的新增代码），应该保持原有的风格。

5 重构、修改部分代码时，应保持和原有代码的命名风格一致。

说明：根据源代码现有的风格继续编写代码，有利于保持总体一致，重构不是另起炉灶，是项目优化。

6 文件名统一采用小写字符。

说明：因为不同系统对文件名大小写处理会不同（如 MS 的 DOS、Windows 系统不区分大小写，但是 Linux 系统则区分），所以代码文件命名建议统一采用全小写字母命名。

7 禁止使用匈牙利命名法

说明：变量命名需要说明的是变量的含义，而不是变量的类型。在变量命名前增加类型说明，反而降低了变量的可读性；更麻烦的问题是，如果修改了变量的类型定义，那么所有使用该变量的地方都需要修改。

8 使用名词或者形容词＋名词方式命名变量

9 函数命名应以函数要执行的动作命名，一般采用动词或者动词＋名词的结构

10 函数指针除了前缀，其他按照函数的命名规则命名

4.变量规范

a 原则

1 一个变量只有一个功能，不能把一个变量用作多种用途。

说明：一个变量只用来表示一个特定功能，不能把一个变量作多种用途，即同一变量取值不同时，其代表的意义也不同。

2 结构功能单一，不设计面面俱到的数据结构。

说明：相关的一组信息才是构成一个结构体的基础，结构的定义应该可以明确的描述一个对象，而不是一组相关性不强的数据的集合。

设计结构应力争使结构代表一种事物抽象，而不是同时代表多种。结构中的各元素应代表同一事物的不同侧面，而不应把相关性不强的不同事物元素放到同一结构中。

3 不用或少用全局变量。

说明：单个文件内部可以使用 static 的全局变量，可以将其理解为类的私有成员变量。

全局变量应是模块私有数据，不能作为对外接口使用，使用 static 类型定义，可以有效防止外部的非正常访问，建议定义 STATIC 宏，在调试阶段，将 STATIC 定义为 static，版本发布时，改为空，以便于后续的打补丁等操作。

b 规则

1 防止局部变量与全局变量同名。

说明：尽管局部变量和全局变量的作用域不同而不会发生语法错误，但容易使人误解。

2 通讯过程中使用的结构，必须注意字节序。

说明：通讯报文中，字节序是一个重要的问题，设备使用的 CPU 类型复杂多样，大小端、32 位/64 位的处理器也都有，如果结构会在报文交互过程中使用，必须考虑字节序问题。位域在不同字节序下，表现看起来差别更大，需要注意。

对于跨平台交互，数据成员发送前，都应该进行主机序到网络序的转换；接收时，也必须进行网络序到主机序的转换。

3 严禁使用未经初始化的变量作为右值。

说明：坚持“在首次使用前初始化变量，初始化的地方离使用的地方越近越好。”这一原则，可以有效避免未初始化错误。

注：部分 IDE 会自动将未初始化变量赋值为 0，但是为了保证安全，开发者自己要杜绝未初始化化错误。

c 建议

1 构造仅有一个模块或函数可以修改、创建，而其余有关模块或函数只访问的全局变量，防止多个不同模块或函数都可以修改、创建同一全局变量的现象。

说明：降低全局变量耦合度。

2 使用面向接口思想，通过 API 访问数据：如果本模块的数据需要对外部模块开放，应提供接口函数来设置、获取，同时注意全局数据的访问互斥。

说明：避免直接暴露内部数据给外部模型使用，是防止模块间耦合最简单有效的方法，注意定义的接口应有较明确的意义。

3 在首次使用前初始化变量，初始化的地方离使用的地方越近越好。

说明：未初始化变量是 C 和 C++程序中错误的常见来源。在变量首次使用前确保正确初始化。在较好的方案中，变量的定义和初始化要做到亲密无间。

4 明确全局变量的初始化顺序，避免跨模块的初始化依赖。

说明：系统启动阶段，使用全局变量前，要考虑到该全局变量在什么时候初始化，使用全局变量和初始化全局变量，两者之间的时序关系，谁先谁后，一定要分析清楚，不然后果往往是低级而又灾难性的。

5 尽量减少没有必要的数据类型默认转换与强制转换。

说明：当进行数据类型强制转换时，其数据的意义、转换后的取值等都有可能发生变化，而这些细节 若考虑不周，就很有可能留下隐患。

示例：将负数转换为无符号数或截断等

5.宏、常量规范

b 规则

1 用宏定义表达式时，要使用完备的括号。

说明：因为宏只是简单的代码替换，不会像函数一样先将参数计算后，再传递。

2 将宏所定义的多条表达式放在大括号中。

说明：更好的方法是多条语句写成 do while(0)的方式，用 do-while(0)方式定义宏，完全不用担心使用者如何使用宏，也不用给使用者加什么约束。

```
示例：#define FOO(x) \                    #define FOO(x) { \
printf("arg is %d\n", x); \      ->       printf("arg is %s\n", x); \
do_something_useful(x);                   do_something_useful(x); \
                                          }

          #define FOO(x) do { \
->        printf("arg is %s\n", x); \
          do_something_useful(x); \
          } while(0)
```

3 使用宏时，不允许参数发生变化。

4 禁止直接使用魔鬼数字。

说明：使用魔鬼数字会造成代码难以理解；如果一个有含义的数字多处使用，一旦需要修改数值，代价惨重。使用明确的物理状态或物理意义的名称能增加信息，并能提供单一的维护点。

注：对于局部使用的唯一含义的魔鬼数字，可以在代码周围增加说明注释，也可以定义局部 const 变量，变量命名自注释。 对于广泛使用的数字，必须定义 const 全局变量/宏，同样变量/宏命名应是自注释的。

且 0 作为一个特殊的数字，作为一般默认值使用没有歧义时，不用特别定义。

c 建议

1 除非必要，应尽可能使用函数代替宏。

说明：宏对比函数，有显著缺点：

α.宏缺乏类型检查，不如函数调用检查严格。

β.宏展开可能会产生意想不到的副作用，如#define SQUARE(a) (a) * (a)这样的定义，如果是 SQUARE(i++)，就会导致 i 被加两次；如果是函数调用 double square(double a) {return a * a;}则不会有此副作用。

γ.以宏形式写的代码难以调试难以打断点，不利于定位问题。如果调用的很多，会造成代码空间的浪费，不如函数空间效率高。

注：如果想要提高效率，可以考虑内联，但是这只适用于稳定且高内聚模块

2 常量建议使用 const 定义代替宏。

说明： 尽量用编译器而不用预处理，因为#define 经常被认为好像不是语言本身的一部分。因为在源码进入编译器之前，它会被预处理程序去掉，于是宏不会加入到符号列表中。如果涉及到这个常量的代码在编译时报错,就会很令人费解,这个问题也会出现在符号调试器中，因为同样地，你所写的符号名不会出现在符号列表中。

注：解决方案：不用预处理宏，定义一个常量。很有效，但有两个特殊情况要注意。首先，定义指针常量时会有点不同。因为常量定义一般是放在头文件中（许多源文件会包含它），除了指针所指的类型要定义成 const 外，重要的是指针也经常要定义成 const。

3 宏定义中尽量不使用 return、goto、continue、break 等改变程序流程的语句。

说明：如果在宏定义中使用这些改变流程的语句，很容易引起资源泄漏问题，开发者很难察觉。

6.质量保证

a 原则

1 代码质量保证优先原则

① 正确性，指程序要实现设计要求的功能。
② 简洁性，指程序易于理解并且易于实现。
③ 可维护性，指程序被修改的能力，包括纠错、改进、新需求或功能规格变化的适应能力。
④ 可靠性，指程序在给定时间间隔和环境条件下，按设计要求成功运行程序的概率。
⑤ 代码可测试性，指软件发现故障并隔离、定位故障的能力，以及在一定的时间和成本前提下，进行测试设计、测试执行的能力。
⑥ 代码性能高效，指是尽可能少地占用系统资源，包括内存和执行时间。
⑦ 可移植性，指为了在原来设计的特定环境之外运行，对系统进行修改的能力。

2 要时刻注意易混淆的操作符。

说明：包括易混淆和的易用错操作符

示例：

除操作符"/" -当除操作符“/”的运算量是整型量时，运算结果也是整型。

求余操作符"%"-求余操作符"%"的运算量只能是整型。

自加、自减操作符“++”、“--”

3 必须了解编译系统的内存分配方式，特别是编译系统对不同类型的变量的内存分配规则，如局部变量在何处分配、静态变量在何处分配等。

4 不仅关注接口，同样要关注实现。

说明：函数所能实现的功能，除了和调用者传递的参数相关，往往还受制于其他隐含约束，如：物理内存的限制，网络状况。

b 规则

1 禁止内存操作越界。

说明：内存操作主要是指对数组、指针、内存地址等的操作。内存操作越界是软件系统主要错误之一，后果往往非常严重，所以进行这些操作时一定要仔细小心。 分析时可以使用内存分析工具检查。

注：坚持下列措施可以避免内存越界：

① 数组的大小要考虑最大情况，避免数组分配空间不够。
② 避免使用危险函数 sprintf /vsprintf/strcpy/strcat/gets 操作字符串，使用相对安全的函数 snprintf/strncpy/strncat/fgets 代替。（这种情况 VS 本身会抛出警告）
③ 使用 memcpy/memset 时一定要确保长度不要越界
④ 字符串考虑最后的’\0’，确保所有字符串是以’\0’结束
⑤ 指针加减操作时，考虑指针类型长度，数组下标进行检查
⑥ 使用时 sizeof 或者 strlen 计算结构/字符串长度，避免手工计算

2 禁止内存泄漏。

说明：内存和资源（包括定时器/文件句柄/Socket/队列/信号量/GUI 等各种资源）泄漏是常见的错误。

注：坚持下列措施可以避免内存泄漏：

- 异常出口处检查内存、定时器/文件句柄/Socket/队列/信号量/GUI 等资源是否全部释放
- 删除结构指针时，必须从底层向上层顺序删除
- 使用指针数组时，确保在释放数组时，数组中的每个元素指针是否已经提前被释放了 避免重复分配内存
- 小心使用有 return、break 语句的宏，确保前面资源已经释放
- 检查队列中每个成员是否释放

3 禁止引用已经释放的内存空间。

说明：在实际编程过程中，稍不留心就会出现在一个模块中释放了某个内存块，而另一模块在随后的某个时刻又使用了它。要防止这种情况发生。

注：坚持下列措施可以避免引用已经释放的内存空间：

- 内存释放后，把指针置为 NULL；使用内存指针前进行非空判断。
- 耦合度较强的模块互相调用时，一定要仔细考虑其调用关系，防止已经删除的对象被再次使用。
- 避免操作已发送消息的内存。
- 自动存储对象的地址不应赋值给其他的在第一个对象已经停止存在后仍然保持的对象（具有更大作用域的对象或者静态对象或者从一个函数返回的对象）

4 编程时，要防止差 1 错误。

说明：此类错误一般是由于把“<=”误写成“<”或“>=”误写成“>”等造成的，由此引起的后果，很多情况下是很严重的，所以编程时，一定要在这些地方小心。当编完程序后，应对这些操作符进行彻底检查，使用变量时要注意其边界值的情况，一般的将常量作为左值可避免。

5 所有的 if ... else if 结构应该由 else 子句结束 ；switch 语句必须有 default 分支。

c 建议

1 函数中分配的内存，在函数退出之前要释放。

说明：有很多函数申请内存，保存在数据结构中，要在申请处加上注释，说明在何处释放。

2 if 语句尽量加上 else 分支，对没有 else 分支的语句要小心对待。

3 不要滥用 goto 语句。

说明：goto 语句会破坏程序的结构性，所以除非确实需要，最好不使用 goto 语句。

可以利用 goto 语句方面退出多重循环；同一个函数体内部存在大量相同的逻辑但又不方便封装成函数的情况下，譬如反复执行文件操作，对文件操作失败以后的处理部分代码（譬如关闭文件句柄，释放动态申请的内存等等），一般会放在该函数体的最后部分，再需要的地方就 goto 到那里，这样代码反而变得清晰简洁。实际也可以封装成函数或者封装成宏，但是这么做会让代码变得没那么直接明了。

4 时刻注意表达式是否会上溢、下溢。

7.程序效率规范

a 原则

1 在保证软件系统的正确性、简洁、可维护性、可靠性及可测性的前提下，提高代码效率。

说明：不能一味地追求代码效率，而对软件的正确、简洁、可维护性、可靠性及可测性造成影响。例如 if 嵌套中应该把执行概率较大的分支放在前面处理。

注：让一个正确的程序更快速，比让一个足够快的程序正确，要容易得太多。大多数时候，不要把注意力集中在如何使代码更快上，应首先关注让代码尽可能地清晰易读和更可靠。

2 通过对数据结构、程序算法的优化来提高效率。

c 建议

1 将不变条件的计算移到循环体外。

说明：将循环中与循环无关，不是每次循环都要做的操作，移到循环外部执行。

2 对于多维大数组，避免来回跳跃式访问数组成员。

3 创建资源库，以减少分配对象的开销。

示例：使用线程池机制，避免线程频繁创建、销毁的系统调用；使用内存池，对于频繁申请、释放的小块内存，一次性申请一个大块的内存，当系统申请内存时，从内存池获取小块内存，使用完毕再释放到内存池中，避免内存申请释放的频繁系统调用.

4 将多次被调用的“小函数”改为 inline 函数或者宏实现。

说明： 如果编译器支持 inline，可以采用 inline 函数。否则可以采用宏。

实现这种优化时要注意 inline 函数优点：其一编译时不用展开，代码 SIZE 小。其二可以加断点，易于定位问题。其三函数编译时，编译器会做语法检查。

8. 注释规范

a 原则

1 优秀的代码可以自我解释，不通过注释即可轻易读懂。

说明：优秀的代码不写注释也可轻易读懂，注释无法把糟糕的代码变好，需要很多注释来解释的代码，往往是失败的，需要重构。

2 注释的内容要清楚、明了，含义准确，防止注释二义性。

说明：有歧义的注释会导致维护者更难看懂代码。

3 在代码的功能、意图层次上进行注释，即注释解释代码难以直接表达的意图，而不是重复描述代码。

说明：注释的目的是解释代码的目的、功能和采用的方法，提供代码以外的信息，帮助读者理解代码，防止没必要的重复注释信息。 对于实现代码中巧妙的、晦涩的、有趣的、重要的地方加以注释。即注释不是为了名词解释（what），而是说明用途（why）。

规则

1 修改代码时，维护代码周边的所有注释，以保证注释与代码的一致性。不再有用的注释要删除。

说明：不要将无用的代码留在注释中，随时可以从源代码配置库中找回代码；即使只是想暂时排除代码，也要留个标注，不然可能会忘记处理它。

2 文件头部应进行注释，注释必须列出：版权说明、版本号、生成日期、开发者者姓名、内容、功能说明、与其它文件的关系、修改日志等，头文件的注释中还应有函数功能简要说明。

说明：通常头文件要对功能和用法作简单说明，源文件包含了更多的实现细节或算法讨论。

3 函数声明处注释描述函数功能、性能及用法，包括输入和输出参数、函数返回值、可重入的要求等；定义处详细描述函数功能和实现要点，如实现的简要步骤、实现的理由、设计约束等。

说明：重要的、复杂的函数，提供外部使用的接口函数应编写详细的注释。

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

5 注释应放在其代码上方相邻位置或右方，不可放在下面。如放于上方则需与其上面的代码用空行隔开，且与下方代码缩进相同。

6 对于 switch 语句下的 case 语句，如果因为特殊情况需要处理完一个 case 后进入下一个 case 处理，必须在该 case 语句处理完、下一个 case 语句前加上明确的注释。

说明：这样比较清楚程序编写者的意图，有效防止无故遗漏 break 语句。

7 避免在注释中使用缩写，除非是业界通用或子系统内标准化的缩写。

8 同一项目组统一注释风格。

建议

1 避免在一行代码或表达式的中间插入注释。

说明：除非必要，不应在代码或表达中间插入注释，否则容易使代码可理解性变差。

2 注释应考虑程序易读及外观排版的因素，使用的语言若是中、英兼有的，建议多使用中文，除非能用非常流利准确的英文表达。

说明：注释语言不统一，影响程序易读性和外观排版，出于对维护人员的考虑，建议使用中文。

3 文件头、函数头、全局常量变量、类型定义的注释格式采用工具可识别的格式。

说明：采用工具可识别的注释格式，例如 doxygen 格式，方便工具导出注释形成帮助文档。

9.排版与格式规范

规则

1 程序块采用缩进风格编写，每级缩进为 4 个空格。

说明：当前各种编辑器/IDE 都支持 TAB 键自动转空格输入，需要打开相关功能并设置相关功能。编辑器/IDE 如果有显示 TAB 的功能也应该打开，方便及时纠正输入错误，IDE 向导生成的代码可以不用修改。

宏定义、编译开关、条件预处理语句可以顶格（或使用自定义的排版方案，但产品/模块内必须保持一致）。

2 相对独立的程序块之间、变量说明之后必须加空行。

3 一条语句不能过长，如不能拆分需要分行写。一行到底多少字符换行比较合适，项目自行确定。

说明：对于目前大多数的 PC 来说，132 比较合适（80/132 是 VTY 常见的行宽值）。注：换行有如下建议：

- 换行时要增加一级缩进，使代码可读性更好；
- 低优先级操作符处划分新行；换行时操作符应该也放下来，放在新行首；
- 换行时建议一个完整的语句放在一行，不要根据字符数断行

4 多个短语句（包括赋值语句）不允许写在同一行内，即一行只写一条语句。

5 if、for、do、while、case、switch、default 等语句独占一行。

说明：执行语句必须用缩进风格写，属于 if、for、do、while、case、switch、default 等下一个缩进级别；

一般写 if、for、do、while 等语句都会有成对出现的“{}”，对此有如下建议可以参考：

- if、for、do、while 等语句后的执行语句建议增加成对的“{}”；
- 如果 if/else 配套语句中有一个分支有“{}”，那么令一个分支即使一行代码也建议增加“{}”；
- 添加“{”的位置可以在 if 等语句后，也可以独立占下一行；独立占下一行时，推荐和 if 在一个缩进级别，也可以在下一个缩进级别；但是如果 if 语句很长，或者已经有换行，建议“{”使用独占一行的写法。

6 在两个以上的关键字、变量、常量进行对等操作时，它们之间的操作符之前、之后或者前后要加空格；进行非对等操作时，如果是关系密切的立即操作符（如->），后不应加空格。

说明：采用这种松散方式编写代码的目的是使代码更加清晰。

在已经非常清晰的语句中没有必要再留空格，如括号内侧(即左括号后面和右括号前面)不需要加空格，多重括号间不必加空格，因为在 C 语言中括号已经是最清晰的标志了。

在长语句中，如果需要加的空格非常多，那么应该保持整体清晰，而在局部不加空格。给操作符留空格时不要连续留两个以上空格。

示例：

- 逗号、分号只在后面加空格
- 比较操作符，赋值操作符"="、"+="，算术操作符"+"、"%"，逻辑操作符"&&"、"&"，位域操作符 "<<"、"^"等双目操作符的前后加空格。
- "!"、"~"、"++"、"--"、"&"（地址操作符）等单目操作符前后不加空格。
- "->"、"."前后不加空格。
- if、for、while、switch 等与后面的括号间应加空格，使 if 等关键字更为突出、明显。

建议

1 注释符（包括“/*”“//”“*/”）与注释内容之间要用一个空格进行分隔。

说明：这样可以使注释的内容部分更清晰，现在很多工具都可以批量生成、删除'//'注释，这样有空格也比较方便统一处理。

2 源程序中关系较为紧密的代码应尽可能相邻。

10.表达式规范

b 规则

1 表达式的值在标准所允许的任何运算次序下都应该是相同的。

说明：除了少数操作符（函数调用操作符（ ）、&&、||、？：和，（逗号）） 之外，子表达式所依据的运算次序是未指定的并会随时更改。注意，运算次序的问题不能使用括号来解决，因为这不是优先级的问题。

将复合表达式分开写成若干个简单表达式，明确表达式的运算次序，就可以有效消除非预期副作用。

示例：

- 自增或自减操作符：表达式会产生不同的结果，把自增运算做为单独的语句，可以避免这个问题。
- 函数参数：函数参数通常从右到左压栈，但函数参数的计算次序不一定与压栈次序相同。
- 函数指针：函数参数和函数自身地址的计算次序未定义
- 嵌套赋值语句：表达式中嵌套的赋值可以产生附加的副作用。不给这种能导致对运算次序的依赖提供任何机会的最好做法是，不要在表达式中嵌套赋值语句。
- volatile 访问：限定符 volatile 表示可能被其它途径更改的变量，例如硬件自动更新的寄存器

c 建议

1 函数调用不要作为另一个函数的参数使用，否则对于代码的调试、阅读都不利。

说明：当函数作为参数时，由于参数压栈次数不是代码可以控制的，可能造成未知的输出，不要为了节约代码行，而写这种代码。

2 赋值语句不要写在 if 等语句中，或者作为函数的参数使用。

说明：因为 if 语句中，会根据条件依次判断，如果前一个条件已经可以判定整个条件，则后续条件语句不会再运行，所以可能导致期望的部分赋值没有得到运行。

3 用括号明确表达式的操作顺序，避免过分依赖默认优先级。

说明：使用括号强调所使用的操作符，防止因默认优先级与设计思想不符而导致程序出错，同时使代码更为清晰可读，但过多的括号会分散代码使其降低了可读性。

注：下面是使用括号的建议：

- 一元操作符，不需要使用括号
- 二元以上操作符，如果涉及多种操作符，则应该使用括号
- 即使所有操作符都是相同的，如果涉及类型转换或者量级提升，也应该使用括号控制计算的次序

4 赋值操作符不能使用在产生布尔值的表达式上。

说明：如果布尔值表达式需要赋值操作，那么赋值操作必须在操作数之外分别进行。这可以帮助避免=和= =的混淆，帮助我们静态地检查错误。

11.代码编辑、编译规范

规则

1 使用编译器的最高告警级别，理解所有的告警，通过修改代码而不是降低告警级别来消除所有告警。

说明：编译器是你的朋友，如果它发出某个告警，这经常说明你的代码中存在潜在的问题。

2 在项目组中，要统一编译开关、静态检查选项以及相应告警清除策略。

说明：如果必须禁用某个告警，应尽可能单独局部禁用，并且编写一个清晰的注释，说明为什么屏蔽。某些语句经编译/静态检查产生告警，但如果你认为它是正确的，那么应通过某种手段去掉告警信息。

3 本地构建工具（如 PC-Lint）的配置应该和持续集成的一致。

说明：两者一致，避免经过本地构建的代码在持续集成上构建失败。

4 使用版本控制（配置管理）系统，及时签入通过本地构建的代码，确保签入的代码不会影响构建成功。

说明：及时签入代码降低集成难度。

建议

1 要小心地使用编辑器提供的块拷贝功能编程。

（三）C++

该规范下紧跟条目的为规则要求，其后的说明词条仅为补充释义便于理解，优点、缺点词条是从正反面补充规则/结论优缺点，对项目组制定项目规范提供支撑。

1.头文件

1.1. Self-contained 头文件

为了保证用户和重构工具不需要为特别场合而包含额外的头文件，头文件应该能够自包含（self-contained,也就是可以作为第一个头文件被引入），以.h 结尾。至于用来插入文本的文件，本质并不是头文件，所以应以 .inc 结尾，不允许分离出-inl.h 头文件的做法。

但当一个文件并不是自包含的，而是作为文本插入到代码某处，或文件内容实际上是其它头文件的特定平台（platform-specific）扩展部分。这些文件就要用.inc 扩展名。

如果 .h 文件声明了一个模板或内联函数，同时也在该文件加以定义。凡是有用到这些的 .cc 文件，就得包含该头文件，否则程序可能会在构建中链接失败。

例外：如果某函数模板为所有相关模板参数显式实例化，或本身就是某类的一个私有成员，那么它就只能定义在实例化该模板的 .cc 文件里。

1.2. #define 保护

所有头文件都应该有 #define 保护来防止头文件被多重包含，命名格式当是: <PROJECT>_<FILE>_H_。

为保证唯一性，头文件的命名应该基于所在项目源代码. 例如，项目 foo 中的头文件 baz.h 可按如下方式保护:

```
#ifndef FOO_BAR_BAZ_H_

#define FOO_BAR_BAZ_H_

...

#endif // FOO_BAR_BAZ_H_
```

1.3. 前置声明

应尽可能避免使用前置声明，尤其是那些定义在其他项目中的实体，使用 #include 包含需要的头文件即可。对于函数只能使用 #include，对于类模板要优先考虑 #include。

说明：所谓「前置声明」（forward declaration）是类、函数和模板的纯粹声明，没伴随着其定义.

优点：

- 前置声明能够节省编译时间，多余的 #include 会迫使编译器展开更多的文件，处理更多的输入。
- 前置声明能够节省不必要的重新编译的时间。#include 使代码因为头文件中无关的改动而被重新编译多次。

缺点：

- 前置声明隐藏了依赖关系，头文件改动时，用户的代码会跳过必要的重新编译过程。
- 前置声明可能会被库的后续更改所破坏。前置声明函数或模板有时会妨碍头文件开发者变动其 API. 例如扩大形参类型，加个自带默认参数的模板形参等等。
- 前置声明来自命名空间 std:: 的 symbol 时，其行为未定义。

- 很难判断什么时候该用前置声明，什么时候该用 #include 。极端情况下，用前置声明代替 #include 甚至都会暗暗地改变代码的含义。
- 前置声明了不少来自头文件的 symbol 时，就会比单单一行的 include 冗长。
- 仅仅为了能前置声明而重构代码（比如用指针成员代替对象成员）会使代码变得更慢更复杂.

注：前置声明的类是不完全类型（incomplete type），只能定义指向该类型的指针或引用，或声明（但不能定义）以不完全类型作为参数或者返回类型的函数，不能创建其类的任何对象，也不能声明成类内部的数据成员。

1.4. 内联函数

- 一个较合理的经验是，不要内联超过 10 行的函数.
- 谨慎对待析构函数，析构函数往往比表面看起来更长，因为有隐含的成员和基类析构函数被调用!
- 不推荐内联那些包含循环或 switch 语句的函数。(除非在大多数情况下，这些循环或 switch 语句从不被执行).
- 有些函数即使声明为内联的也不一定会被编译器内联,比如虚函数和递归函数就不会被正常内联.通常，递归函数不应该声明成内联函数.

注：递归调用堆栈的展开并不像循环那么简单,比如递归层数在编译时可能是未知的，大多数编译器都不支持内联递归函数. 虚函数内联的主要原因则是想把它的函数体放在类定义内，为了图个方便，抑或是当作文档描述其行为，比如精短的存取函数.

说明:当函数被声明为内联函数之后，编译器会将其内联展开，而不是按通常的函数调用机制进行调用.

优点:只要内联的函数体较小，内联该函数可以令目标代码更加高效. 对于存取函数以及其它函数体比较短，性能关键的函数，鼓励使用内联.

缺点:滥用内联将导致程序变得更慢. 内联可能使目标代码量或增或减，这取决于内联函数的大小. 内联非常短小的存取函数通常会减少代码大小，但内联一个相当大的函数将戏剧性的增加代码大小. 现代处理器由于更好的利用了指令缓存，小巧的代码往往执行更快。

注：类内部的函数一般会自动内联。所以某函数一旦不需要内联，定义就不要再放在头文件里，而是放到对应的 .cc 文件里。这样可以保持头文件的类精炼，也符合声明与定义分离的原则。

1.5. #include 的路径及顺序

使用标准的头文件包含顺序可增强可读性，避免隐藏依赖：相关头文件，C 库，C++ 库，其他库的 .h，本项目内的 .h.这种优先的顺序排序保证当遗漏某些必要的库时，程序的构建会立刻中止。

建议：

- 项目内头文件应按照项目源代码目录树结构排列，避免使用 UNIX 特殊的快捷目录 .（当前目录）或 ..（上级目录）.

- 可按字母顺序分别对每种类型的头文件进行二次排序。
- 将库文件放在最后，让出错的先是项目内文件，头文件都放在对应源文件的最前面，足以保证内部错误及时发现，即按不稳定顺序包含。
- 所依赖的符号（symbols）被哪些头文件所定义，就应该包含（include）哪些头文件，前置声明除外。
- 如平台特定（system-specific）代码需要条件编译（conditional includes），代码可以放到其它 includes 之后。
- 在 #include 中插入空行或注释以分割相关头文件，C 库，C++ 库，其他库的 .h 和本项目内的 .h 是个好习惯。

示例：要用 bar.h 中的某个标识符，哪怕所包含的 foo.h 已包含了 bar.h，也应该照样包含 bar.h，除非 foo.h 有明确说明它会自动向程序提供 bar.h 中的 symbol. 不过，凡是 cc 文件所对应的「相关头文件」已经包含的，就不用再重复包含进其 cc 文件里面了，例如 foo.cc 只包含 foo.h 就够了，不用管后者所包含的其它内容。

2.作用域

作用域的使用，除了考虑名称污染，可读性之外，主要是为降低耦合，提高编译/执行效率.

2.1. 命名空间

- 推荐在 .cc 文件内使用匿名命名空间或 static 声明. 使用具名的命名空间时，其名称可基于项目名或相对路径.
- 根据下文将要提到的策略合理使用命名空间.
- 遵守“命名空间命名”中的规则。
- 在命名空间的最后注释出命名空间的名字。
- 用命名空间把文件包含以及类的前置声明以外的整个源文件封装起来，以区别于其它命名空间。
- 禁止在命名空间 std 内声明任何东西，包括标准库的类前置声明. 在 std 命名空间声明实体是未定义的行为，会导致不可移植.声明标准库下的实体，需要包含对应的头文件.
- 禁止使用 using 指示引入整个命名空间的标识符号，这会污染命名空间
- 不要在头文件中使用命名空间别名 除非显式标记内部命名空间使用。因为任何在头文件中引入的命名空间都会成为公开 API 的一部分。
- 禁止用内联命名空间

说明：命名空间将全局作用域细分为独立的，具名的作用域，可有效防止全局作用域的命名冲突.

优点:

- 虽然类已经提供了（可嵌套的）命名轴线（注：将命名分割在不同类的作用域内），命名空间在这基础上又封装了一层.
- 两个不同项目的全局作用域都有一个类，这样在编译或运行时造成冲突. 如果每个项目将代码置于不同命名空间中，就不会冲突.
- 内联命名空间会自动把内部的标识符放到外层作用域

缺点：

- 命名空间具有迷惑性，因为它们使得区分两个相同命名所指代的定义更加困难。
- 内联命名空间很容易令人迷惑，毕竟其内部的成员不再受其声明所在命名空间的限制。内联命名空间只在大型版本控制里有用。
- 有时候不得不多次引用某个定义在许多嵌套命名空间里的实体，使用完整的命名空间会导致代码的冗长。
- 在头文件中使用匿名空间导致违背 C++ 的唯一定义原则 (One Definition Rule (ODR))。

2.2. 匿名命名空间和静态变量

- 在 .cc 文件中定义一个不需要被外部引用的变量时使用内部链接性声明（放在匿名命名空间或声明为 static），但是不要在 .h 中使用。
- 匿名命名空间的声明和具名的格式相同，在最后注释上 namespace

说明:所有置于匿名命名空间的声明都具有内部链接性，函数和变量可以经由声明为 static 拥有内部链接性，这意味着你在这个文件中声明的这些标识符都不能在另一个文件中被访问。即使两个文件声明了完全一样名字的标识符，它们所指向的实体实际上是完全不同的。

2.3. 非成员函数、静态成员函数和全局函数

使用静态成员函数或命名空间内的非成员函数，尽量不要用裸的全局函数. 将一系列函数直接置于命名空间中，不要用类的静态方法模拟出命名空间的效果，类的静态方法应当和类的实例或静态数据紧密相关.

- 有时，把函数的定义同类的实例脱钩是有益的，这样的函数可以被定义成静态成员，或是非成员函数. 非成员函数不应依赖于外部变量，应尽量置于某个命名空间内. 相比单纯为了封装若干不共享任何静态数据的静态成员函数而创建类，不如使用 2.1 命名空间。
- 定义在同一编译单元的函数，被其他编译单元直接调用可能会引入不必要的耦合和链接时依赖；静态成员函数对此尤其敏感. 可以考虑提取到新类中，或者将函数置于独立库的命名空间内.
- 如果你必须定义非成员函数，又只是在 .cc 文件中使用它，可使用匿名 2.1 命名空间或 static 链接关键字 (如 static int Foo() {...}) 限定其作用域.

优点:某些情况下，非成员函数和静态成员函数是非常有用的，将非成员函数放在命名空间内可避免污染全局作用域.

缺点:将非成员函数和静态成员函数作为新类的成员或许更有意义，当它们需要访问外部资源或具有重要的依赖关系时更是如此.

2.4. 局部变量

- 将函数变量尽可能置于最小作用域内，并在变量声明时进行初始化.C++允许在函数的任何位置声明变量. 推荐在尽可能小的作用域中声明变量，离第一次使用越近越好.这使得代码浏览者更容易定位变量声明的位置,了解变量的类型和初始值. 特别是，应使用初始化的方式替代声明再赋值,

- 属于 if, while 和 for 语句的变量应当在这些语句中正常地声明，这样子这些变量的作用域就被限制在这些语句中了

例外：如果变量是一个对象，每次进入作用域都要调用其构造函数，每次退出作用域都要调用其析构函数. 这会导致效率降低.在循环作用域外面声明这类变量要高效的多。

2.5. 静态和全局变量

- 禁止定义静态储存周期非 POD(POD : Plain Old Data-原生数据类型)变量，禁止使用含有副作用的函数初始化 POD 全局变量,因为多编译单元中的静态变量执行时的构造和析构顺序是未明确的，这将导致代码的不可移植。
- 全局变量，静态变量，静态类成员变量和函数静态变量，都必须是 POD 类型以及 POD 类型的指针、数组和结构体。
- 禁止使用类的静态储存周期变量。由于构造和析构函数调用顺序的不确定性，它们会导致难以发现的 bug。constexpr 变量除外，因为不涉及动态初始化或析构。
- 只允许 POD 类型的静态变量，完全禁用 vector（使用 C 数组替代）和 string（使用 const char []）。
- 确实需要一个 class 类型的静态或全局变量，可以考虑在 main() 函数或 pthread_once() 内初始化一个指针且永不回收。并且只能用 raw 指针，禁止使用智能指针，因为后者析构函数涉及到上文指出的不定顺序问题。

静态变量的构造函数、析构函数和初始化的顺序在 C++ 中是只有部分明确的，甚至随着构建变化而变化，导致难以发现的 bug. 所以除了禁用类类型的全局变量，我们也不允许用函数返回值来初始化 POD 变量，除非该函数（比如 getenv() 或 getpid() ）不涉及任何全局变量。函数作用域里的静态变量除外，毕竟它的初始化顺序是有明确定义的，而且只会在指令执行到它的声明那里才会发生。

注:同一个编译单元内是明确的，静态初始化优先于动态初始化，初始化顺序按照声明顺序进行，销毁则逆序。不同的编译单元之间初始化和销毁顺序属于未明确行为。

同理，全局和静态变量在程序中断时会被析构，无论中断是从 main() 返回还是对 exit() 的调用。析构顺序正好与构造函数调用的顺序相反，但既然构造顺序未定义，那么析构顺序当然也就不确定了。比如在程序结束时某静态变量已经被析构了，但代码还在跑如：其它线程试图访问它且失败,或者一个静态 string 变量也许会在一个引用了前者的其它变量析构之前被析构掉。

改善办法：用 quick_exit() 来代替 exit() 并中断程序。前者不会执行任何析构，也不会执行 atexit() 所绑定的任何 handlers. 如果想在执行 quick_exit() 来中断时执行某 handler（比如刷新 log），可以把它绑定到 _at_quick_exit(). 如果想在 exit() 和 quick_exit() 都用上该 handler, 都绑定上去。

注:

- 上文提及的静态变量泛指静态生存周期的对象，包括：全局变量，静态变量，静态类成员变量，以及函数静态变量.

- 尽量不用全局函数和全局变量，考虑作用域和命名空间限制，尽量单独形成编译单元；
- 多线程中的全局变量（含静态成员变量）不要使用 class 类型（含 STL 容器），避免不明确行为导致的 bug.

3. 类

3.1. 构造函数的职责

构造函数可以进行初始化操作，但禁止在构造函数中调用虚函数，同时禁止在无法报出错误时进行可能失败的初始化.如果代码允许，直接终止程序是一个合适的处理错误的方式. 否则，考虑用 Init() 方法或工厂函数（factory function，出自 C++ 的一种设计模式，即简单工厂模式）.如果对象需要进行有意义的 (non-trivial) 初始化，也考虑使用明确的 Init()方法或工厂模式。

优点

- 无需考虑类是否被初始化.
- 经过构造函数完全初始化后的对象可以为 const 类型，也能更方便地被标准容器或算法使用.

缺点

- 如果在构造函数内调用了自身的虚函数，这类调用是不会重定向到子类的虚函数实现. 即使当前没有子类化实现，将来仍是隐患.
- 在没有使程序崩溃 (因为并不是一个始终合适的方法) 或者使用异常 (因为已经被禁用了) 等方法的条件下，构造函数很难上报错误
- 如果执行失败，会得到一个初始化失败的对象，这个对象有可能进入不正常的状态，必须使用 bool IsValid() 或类似这样的机制才能检查出来，然而这是一个十分容易被疏忽的方法.
- 构造函数的地址是无法被取得的，因此，举例来说，由构造函数完成的工作是无法以简单的方式交给其他线程的.

3.2. 隐式类型转换

- 不要定义隐式类型转换. 对于转换运算符和单参数构造函数，使用 explicit 关键字.在类型定义中，类型转换运算符和单参数构造函数都应当用 explicit 进行标记。
- 不能以一个参数进行调用的构造函数不应当加上 explicit.
- 接受一个 std::initializer_list 作为参数的构造函数也应当省略 explicit，以便支持拷贝初始化

例外： 拷贝和移动构造函数不应当被标记为 explicit，因为它们并不执行类型转换. 对于设计目的就是用于对其他类型进行透明包装的类来说，隐式类型转换有时是必要且合适的. 这时应当联系项目组长并说明特殊情况.

说明：

- 隐式类型转换允许一个某种类型（称作 源类型）的对象被用于需要另一种类型（称作目的类型）的位置，例如，将一个 int 类型的参数传递给需要 double 类型的函数.
- 除了语言所定义的隐式类型转换，用户还可以通过在类定义中添加合适的成员定义自己需要的转换. 在源类型中定义隐式类型转换，可以通过目的类型名的类型转换运算符实现（例如 operator bool()）. 在目的类型中定义隐式类型转换，则通过以源类型作为其唯一参数（或唯一无默认值的参数）的构造函数实现.
- explicit 关键字可以用于构造函数或类型转换运算符（C++11 引入），以保证只有当目的类型在调用点被显式写明时才能进行类型转换，例如使用 cast. 这不仅作用于隐式类型转换，还能作用于 C++11 的列表初始化语法。

优点

- 有时目的类型名是一目了然的，通过避免显式地写出类型名，隐式类型转换可以让一个类型的可用性和表达性更强.
- 隐式类型转换可以简单地取代函数重载.
- 在初始化对象时，列表初始化语法是一种简洁明了的写法.

缺点

- 隐式类型转换会隐藏类型不匹配的错误. 有时，目的类型并不符合用户的期望，甚至用户根本没有意识到发生了类型转换.
- 隐式类型转换会让代码难以阅读，尤其是在有函数重载的时候，因为这时很难判断到底是哪个函数被调用.
- 单参数构造函数有可能会被无意地用作隐式类型转换.
- 如果单参数构造函数没有加上 explicit 关键字，读者无法判断这一函数究竟是要作为隐式类型转换，还是作者忘了加上 explicit 标记.
- 并没有明确的方法用来判断哪个类应该提供类型转换，这会使得代码变得含糊不清.
- 如果目的类型是隐式指定的，那么列表初始化会出现和隐式类型转换一样的问题，尤其是在列表中只有一个元素的时候.

3.3. 可拷贝类型和可移动类型

- 如果需要就让类型可拷贝/可移动. 作为经验——如果对于用户来说这个拷贝操作不是一眼就能看出来的，就不要把类型设置为可拷贝. 如果让类型可拷贝，一定要同时给出拷贝构造函数和赋值操作的定义，反之亦然. 如果让类型可拷贝，同时移动操作的效率高于拷贝操作，那么就把移动的两个操作（移动构造函数和赋值操作）也给出定义. 如果类型不可拷贝，但是移动操作的正确性对用户显然可见，那么把这个类型设置为只可移动并定义移动的两个操作.

- 如果定义了拷贝/移动操作，则要保证这些操作的默认实现是正确的. 记得时刻检查默认操作的正确性，并且在文档中说明类是可拷贝的且/或可移动的.
- 由于存在对象切割的风险，不要为任何有可能有派生类的对象提供赋值操作或者拷贝 / 移动构造函数（当然也不要继承有这样的成员函数的类）. 如果你的基类需要可复制属性，请提供一个 public virtual Clone() 和一个 protected 的拷贝构造函数以供派生类实现.
- 如果你的类不需要拷贝 / 移动操作，请显式地通过在 public 域中使用 = delete 或其他手段禁用之.

注：

- 可拷贝类型允许对象在初始化时得到来自相同类型的另一对象的值，或在赋值时被赋予相同类型的另一对象的值，同时不改变源对象的值. 对于用户定义的类型，拷贝操作一般通过拷贝构造函数与拷贝赋值操作符定义. string 类型就是一个可拷贝类型的例子.
- 可移动类型允许对象在初始化时得到来自相同类型的临时对象的值，或在赋值时被赋予相同类型的临时对象的值（因此所有可拷贝对象也是可移动的）. std::unique_ptr<int> 就是一个可移动但不可复制的对象的例子. 对于用户定义的类型，移动操作一般是通过移动构造函数和移动赋值操作符实现的.
- 拷贝 / 移动构造函数在某些情况下会被编译器隐式调用. 例如，通过传值的方式传递对象.

优点

- 可移动及可拷贝类型的对象可以通过传值的方式进行传递或者返回，这使得 API 更简单，更安全也更通用. 与指针传递和引用不同，这样的传递不会造成所有权，生命周期，可变性等方面混乱，也就没必要在协议中予以明确. 同时也防止了客户端与实现在非作用域内的交互，使得它们更容易被理解与维护. 这样的对象可以和需要传值操作的通用 API 一起使用。
- 拷贝/移动构造函数与赋值操作一般来说要比它们的各种替代方案，比如 Clone()、 CopyFrom() 、Swap()更容易定义，因为它们能通过编译器产生，无论是隐式的还是通过 = default. 这种方式很简洁，也保证所有数据成员都会被复制. 拷贝与移动构造函数一般也更高效，因为它们不需要堆的分配或者是单独的初始化和赋值步骤，同时，对于类似省略不必要的拷贝这样的优化它们也更加合适.
- 移动操作允许隐式且高效地将源数据转移出右值对象. 这有时能让代码风格更加清晰.

缺点

- 许多类型都不需要拷贝，为它们提供拷贝操作会让人迷惑，也显得荒谬而不合理. 单件类型 (Registerer)，与特定的作用域相关的类型 (Cleanup)，与其他对象实体紧耦合的类型 (Mutex) 从逻辑上来说都不应该提供拷贝操作. 为基类提供拷贝 / 赋值操作是有害的，因为在使用它们时会造成对象切割. 默

认的或者随意的拷贝操作实现可能是不正确的，这往往导致令人困惑并且难以诊断出的错误.

- 拷贝构造函数是隐式调用的，也就是说，这些调用很容易被忽略. 这会让人迷惑，尤其是对那些所用的语言约定或强制要求传引用的程序员来说更是如此. 同时，这从一定程度上说会鼓励过度拷贝，从而导致性能上的问题.

3.4. 结构体/类

仅当只有数据成员时使用 struct，其它一概使用 class.

说明：在 C++ 中 struct 和 class 关键字几乎含义一样. 为这两个关键字添加我们自己的语义理解，以便为定义的数据类型选择合适的关键字.

- struct 用来定义包含数据的被动式对象，也可以包含相关的常量，但除了存取数据成员之外，没有别的函数功能. 并且存取功能是通过直接访问位域，而非函数调用. 除了构造函数，析构函数，Initialize()，Reset()，Validate()等类似的用于设定数据成员的函数外，不能提供其它功能的函数.
- 如果需要更多的函数功能，class 更适合. 如果拿不准,也请用 class.
- 为了和 STL 保持一致，对于仿函数等特性可以不用 class 而是使用 struct.

注:类和结构体的成员变量使用不同的命名规则.

3.5. 继承

- 使用组合常常比使用继承更合理. 如果使用继承的话，定义为 public 继承.
- 所有继承必须是 public 的. 如果使用私有继承，用法应该替换成把基类实例作为成员对象的方式.
- 不要过度使用实现继承.组合常常更合适一些.尽量做到只在 “是一个”的情况下使用继承。例如：Bar 的确“是一种”Foo，Bar 才能继承 Foo.
- 必要的话，析构函数声明为 virtual.如果你的类有虚函数，则析构函数也应该为虚函数.
- 对于可能被子类访问的成员函数，不要过度使用 protected 关键字.
- 对于重载的虚函数或虚析构函数，使用 override，或 final 关键字显式标记. 早于 C++11 的代码会使用 virtual 关键字. 因此，在声明重载时，请使用 override，final 或 virtual 其中之一进行标记. 标记为 override 或 final 的析构函数如果不是对基类虚函数的重载的话，编译会报错，这有助于捕获常见错误. 这些标记也起到了文档的作用，如果省略这些关键字，debug 不得不检查所有父类，以判断该函数是否是虚函数.
- 数据成员都必须是私有的

注：“is-a”用继承,其他 “has-a” 情况下使用组合

说明：当子类继承基类时，子类包含了父基类所有数据及操作的定义. 实践中，继承主要用于两种场合：实现继承，子类继承父类的实现代码；接口继承，子类仅继承父类的方法名称.

优点

实现继承通过原封不动的复用基类代码减少了代码量. 由于继承是在编译时声明, 程序员和编译器都可以理解相应操作并发现错误. 从编程角度而言, 接口继承是用来强制类输出特定的 API. 在类没有实现 API 中某个必须的方法时, 编译器同样会发现并报告错误.

缺点

对于实现继承, 由于子类的实现代码散布在父类和子类间之间, 要理解其实现变得更加困难. 子类不能重写父类的非虚函数, 当然也就不能修改其实现. 基类也可能定义了一些数据成员, 因此还必须区分基类的实际布局.

3.6. 多重继承

真正需要用到多重实现继承的情况少之又少，只在以下情况才允许多重继承:

- 最多只有一个基类是非抽象类;
- 其它基类都是以 Interface 为后缀的纯接口类.

说明：多重继承允许子类拥有多个基类. 要将作为纯接口的基类和具有实现的基类区别开来.

优点

相比单继承, 多重实现继承可以复用更多的代码.

缺点

真正需要用到多重实现继承的情况少之又少. 有时多重实现继承看上去是不错的解决方案, 但通常也可以找到一个更明确, 更清晰的不同解决方案.

3.7. 接口

说明:接口是指满足特定条件的类, 这些类以 Interface 为后缀 (不强制).具体细节可参考 Stroustrup——The C++ Programming Language, 3rd edition 第 12.4 节.

当一个类满足以下要求时, 称之为纯接口:

- 只有纯虚函数 ("=0") 和静态函数 (除了下文提到的析构函数).
- 没有非静态数据成员.
- 没有定义任何构造函数. 如果有, 也不能带有参数, 并且必须为 protected.
- 如果它是一个子类, 也只能从满足上述条件并以 Interface 为后缀的类继承.

注：接口类不能被直接实例化，因为它声明了纯虚函数. 为确保接口类的所有实现可被正确销毁，必须为之声明虚析构函数.

优点：以 Interface 为后缀可以提醒其他人不要为该接口类增加函数实现或非静态数据成员. 这一点对于多重继承尤其重要.

缺点：Interface 后缀增加了类名长度，为阅读和理解带来不便. 同时，接口属性作为实现细节不应暴露给用户.

3.8. 运算符重载

- 除少数条件下，不要重载运算符或创建用户定义字面量.
- 只有在意义明显，不出现奇怪的行为且与对应内建运算符行为一致时定义重载运算符.
- 只有对用户自己定义的类型重载运算符. 即将它们和它们所操作的类型定义在同一个头文件中,.cc 中和命名空间中. 这样做无论类型在哪里都能够使用定义的运算符,并且最大程度上避免了多重定义的风险.不推荐将运算符定义为模板，因为此时它们对任何模板参数都能够作用.如果定义了一个运算符,请将其相关且有意义的运算符都进行定义,并保证定义的语义一致.
- 建议不要将不进行修改的二元运算符定义为成员函数.如果一个二元运算符被定义为类成员,这时隐式转换会作用域右侧的参数却不会作用于左侧.这时会出现 a<b 能够通过编译而 b<a 不能的情况。
- 不要为了避免重载操作符而走极端.不要只是为了满足函数库需要而去定义运算符重载.如果类型没有自然顺序，要将它们存入 std::set 中,最好还是定义一个自定义的比较运算符而不是重载<。
- 不要重载 &&,||或一元运算符 &.不要重载 operator""。不要引入用户定义字面量.

说明：C++ 允许通过使用 operator 关键字对内建运算符进行重载定义,只要其中一个参数是用户定义的类型.operator 关键字还允许用户使用 operator"" 定义新的字面运算符或类型转换函数，例如 operator bool().

优点：重载运算符可以让代码更简洁易懂，也使得用户定义的类型和内建类型拥有相似的行为. 重载运算符对于某些运算来说是符合符合语言习惯的名称（例如 ==, <, =, <<），遵循这些语言约定可以让用户定义的类型更易读，也能更好地和需要这些重载运算符的函数库进行交互操作.对于创建用户定义的类型的对象来说，用户定义字面量是一种非常简洁的标记.

缺点

- 要提供正确，一致，不出现异常行为的操作符运算需要花费不少精力，而且如果达不到这些要求的话，会导致令人迷惑的 Bug.
- 过度使用运算符会带来难以理解的代码，尤其是在重载的操作符的语义与通常的约定不符合时.
- 函数重载有多少弊端，运算符重载就至少有多少.

- 运算符重载会混淆视听，让你误以为一些耗时的操作和操作内建类型一样轻巧.
- 对重载运算符的调用点的查找需要的可就不仅仅是像 grep 那样的程序了，这时需要能够理解 C++ 语法的搜索工具.
- 如果重载运算符的参数写错，此时得到的可能是一个完全不同的重载而非编译错误. 例如: foo < bar 执行的是一个行为，而 &foo < &bar 执行的就是完全不同的另一个行为了.
- 重载某些运算符本身就是有害的. 例如，重载一元运算符 & 会导致同样的代码有完全不同的含义，这取决于重载的声明对某段代码而言是否是可见的. 重载诸如 &&, || 和 ，会导致运算顺序和内建运算的顺序不一致.
- 运算符从通常定义在类的外部，所以对于同一运算，可能出现不同的文件引入了不同的定义的风险. 如果两种定义都链接到同一二进制文件，就会导致未定义的行为，有可能表现为难以发现的运行时错误.
- 用户定义字面量所创建的语义形式对于某些有经验的 C++ 程序员来说都是很陌生的.

3.9. 存取控制

将所有数据成员声明为 private，除非是 static const 类型成员（遵循常量命名规则）.

3.10. 声明顺序

- 将相似的声明放在一起，将 public 部分放在最前.
- 建议将类似的声明放在一起，并且建议以如下顺序：类型（包括 typedef，using 和嵌套的结构体与类），常量，工厂函数，构造函数，赋值运算符，析构函数，其它函数，数据成员.
- 不要将大段的函数定义内联在类定义中. 通常只有那些普通的，或性能关键且短小的函数可以内联在类定义中.

说明：类定义一般应以 public: 开始，后跟 protected:，最后是 private:. 省略空部分.

4. 函数

4.1. 参数顺序

函数的参数顺序为:输入参数在先,输出参数在后，但并非是强制的，只是推荐这么做。

说明：C/C++中的函数参数或是函数的输入、函数的输出，或二者都有.输入参数通常是值参或 const 引用,输出参数或输入/输出参数则一般为非 const 指针.在排列参数顺序时,将所有的输入参数置于输出参数之前.注意在加入新参数时仍然要按照前述的规则,将新的输入参数也置于输出参数之前.

4.2. 编写简短函数

推荐编写简短，凝练的函数.

说明

- 长函数确实是合理的，因此并不硬性限制函数长度.但如果函数超过 40 行,可以思索一下能否在不影响程序结构的前提下进行分割.
- 即使一个长函数现在工作的非常好,一旦有人修改,就有可能出现新问题,甚至导致难以发现的 bug.函数尽量简短，便于他人阅读和修改代码.
- 处理代码时，可能会有复杂的长函数.不要害怕修改现有代码——如果证实这些代码使用/调试起来很困难,或者只需要使用其中的一段,考虑将其分割为更加简短并易于管理的若干函数.

4.3. 引用参数

所有按引用传递的参数必须加上 const.即输入形参是 const T&.如果用 const T*，说明输入另有处理.使用 const T*时应注释出相应的理由,否则会使读者感到迷惑.

说明：C 语言中,如果函数需要修改变量的值，参数必须为指针,如 int foo(int *pval).在 C++中,函数还可以声明为引用参数：int foo(int &val).

优点：定义引用参数可以防止出现 (*pval)++ 这样丑陋的代码. 引用参数对于拷贝构造函数这样的应用也是必需的. 同时也更明确地不接受空指针.

缺点：容易引起误解，因为引用在语法上是值变量却拥有指针的语义.

注：有时在输入形参中用 const T* 指针比 const T& 更好. 原因:

- 可能会传递空指针.
- 函数要把指针或对地址的引用赋值给输入形参.

4.4. 函数重载

要使用函数重载，必须能让读者一看调用点就了然，不用花心思猜测调用的重载函数到底是哪一种，当然这一点也适用于构造函数.

注:打算重载一个函数这方法也可以替换为在函数名中加参数信息.例如,用 AppendString()和 AppendInt(),而非重载多个 Append().如果重载函数的目的是为了支持不同数量的同一类型参数,则优先考虑使用 std::vector 以便使用者可以用列表初始化指定参数.

说明：可以编写一个参数类型为 const string& 的函数，然后用另一个参数类型为 const char* 的函数对其进行重载:

优点:通过重载参数不同的同名函数，可以令代码更加直观. 模板化代码需要重载，这同时也能为使用者带来便利.

缺点：如果函数单靠不同的参数类型而重载,读者就得十分熟悉 C++ 五花八门的匹配规则,以了解匹配过程具体到底如何. 另外，如果派生类只重载了某个函数的部分变体，继承语义就容易令人困惑.

4.5. 缺省参数

- 只允许在非虚函数中使用缺省参数，且必须保证缺省参数的值始终一致. 缺省参数与函数重载 遵循同样的规则. 一般建议使用函数重载,尤其是在缺省函数带来的可读性提升不能弥补下文中所提到的缺点.
- 虚函数不允许使用缺省参数,因为在虚函数中缺省参数不一定能正常工作.如果在每个调用点缺省参数的值都有可能不同,在这种情况下缺省函数也不允许使用.(例如：不要写像 void f(int n = counter++);这样的代码)
- 如果缺省参数对可读性的提升远远超过了以上提及的缺点的话，可以使用缺省参数. 如果仍有疑惑，就使用函数重载.

优点：有些函数一般情况下使用默认参数，但有时需要又使用非默认的参数. 缺省参数为这样的情形提供了便利，使程序员不需要为了极少的例外情况编写大量的函数. 和函数重载相比，缺省参数的语法更简洁明了，减少了大量的样板代码，也更好地区别了“必要参数”和“可选参数”

缺点

- 缺省参数实际上是函数重载语义的另一种实现方式，因此所有 不应当使用函数重载的理由 也都适用于缺省参数.
- 虚函数调用的缺省参数取决于目标对象的静态类型，此时无法保证给定函数的所有重载声明的都是同样的缺省参数.
- 缺省参数是在每个调用点都要进行重新求值的，这会造成生成的代码迅速膨胀. 作为读者，一般来说也更希望缺省的参数在声明时就已经被固定了，而不是在每次调用时都可能会有不同的取值.
- 缺省参数会干扰函数指针，导致函数签名与调用点的签名不一致. 而函数重载不会导致这样的问题.

4.6. 函数返回类型后置语法

- 只有在常规写法(返回类型前置)不便于书写或不便于阅读时使用返回类型后置语法.
- 大部分情况下,应继续使用以往的函数声明写法,将返回类型置于函数名前.只有在必需的时候 (如 Lambda 表达式) 或者使用后置语法能够简化书写并且提高易读性的时候才使用新的返回类型后置语法.但后一种情况一般比较少见,大部分出现在复杂模板代码中,而多数情况下不鼓励写复杂模板代码.

说明：C++现在允许两种不同的函数声明方式：以往写法是将返回类型置于函数名之前;C++11引入了一新形式，可以在函数名前使用 auto 关键字,在参数列表后后置返回类型

后置返回类型为函数作用域. 对于像 int 这样简单的类型，两种写法没有区别. 但对于复杂的情况，例如类域中的类型声明或者以函数参数的形式书写的类型，写法的不同会造成区别.

优点：后置返回类型是显式指定 Lambda 表达式返回值唯一方式.某些情况下,编译器可以自动推导出 Lambda 表达式返回类型,但并不是在所有的情况下都能实现.即使编译器能够自动

推导,显式地指定返回类型也能让读者更明了.有时在已经出现了的函数参数列表之后指定返回类型,能让书写更简单,也更易读,尤其是返回类型依赖于模板参数时.

缺点:后置返回类型相对是非常新的语法,而且在 C 和 Java 中都没有相似写法, 因此可能比较陌生.实际的做法是使用旧的语法或者新旧混用.这种情况下,只使用一种版本是相对更规整的形式.

4.7. 所有权与智能指针

- 动态分配出的对象最好有单一且固定的所有主, 并通过智能指针传递所有权.
- 如果必须使用动态分配, 那么要将所有权保持在分配者手中.如果其他地方要使用这个对象,最好传递它的拷贝,或传递一个不用改变所有权的指针或引用.推荐使用 std::unique_ptr 来明确所有权传递
- 如果不必要,不要使用共享所有权.例如为了避免开销昂贵的拷贝操作,只有当性能提升非常明显,并且操作的对象不可变(例如 std::shared_ptr<const Foo>)时,才能这么做.如果确实要使用共享所有权,推荐使用 std::shared_ptr .
- 不要使用 std::auto_ptr,使用 std::unique_ptr 代替它.

说明:

- 所有权是一种登记/管理动态内存和其它资源的技术. 动态分配对象的所有主是一个对象或函数, 后者负责确保当前者无用时就自动销毁前者. 所有权有时可以共享, 此时就由最后一个所有主来负责销毁它. 甚至也可以不用共享, 在代码中直接把所有权传递给其它对象.
- 智能指针是一个通过重载 * 和 -> 运算符以表现得如指针一样的类. 智能指针类型被用来自动化所有权的登记工作, 来确保执行销毁义务到位. std::unique_ptr 是 C++11 新推出的一种智能指针类型, 用来表示动态分配出的对象的独一无二的所有权; 当 std::unique_ptr 离开作用域时, 对象就会被销毁. std::unique_ptr 不能被复制, 但可以把它移动(move)给新所有主. std::shared_ptr 同样表示动态分配对象的所有权, 但可以被共享, 也可以被复制; 对象的所有权由所有复制者共同拥有, 最后一个复制者被销毁时, 对象也会随着被销毁.

优点

- 如果没有清晰、逻辑条理的所有权安排, 不可能管理好动态分配的内存.
- 传递对象的所有权, 开销比复制来得小, 如果可以复制的话.
- 传递所有权也比"借用"指针或引用来得简单, 毕竟它大大省去了两个用户一起协调对象生命周期的工作.
- 如果所有权逻辑条理, 有文档且不紊乱的话, 可读性会有很大提升.
- 可以不用手动完成所有权的登记工作, 大大简化了代码, 也免去了一大波错误之恼.
- 对于 const 对象来说, 智能指针简单易用, 也比深度复制高效.

缺点

- 不得不用指针（不管是智能的还是原生的）来表示和传递所有权. 指针语义可要比值语义复杂得许多了, 特别是在 API 里:这时不光要操心所有权, 还要顾及别名, 生命周期, 可变性以及其它大大小小的问题.
- 其实值语义的开销经常被高估, 所以所有权传递带来的性能提升不一定能弥补可读性和复杂度的损失.
- 如果 API 依赖所有权的传递, 就会害得客户端不得不用单一的内存管理模型.
- 如果使用智能指针, 那么资源释放发生的位置就会变得不那么明显.
- std::unique_ptr 的所有权传递原理是 C++11 的 move 语法, 后者毕竟是刚刚推出的, 容易迷惑程序员.
- 如果原本的所有权设计已经够完善了, 那么若要引入所有权共享机制, 可能不得不重构整个系统.
- 所有权共享机制的登记工作在运行时进行, 开销可能相当大.
- 某些极端情况下 (例如循环引用), 所有权被共享的对象永远不会被销毁.
- 智能指针并不能够完全代替原生指针.

注：

- 智能指针当成对象来看待, 很好领会它与所指对象之间的关系
- Rust 的 Ownership 思想是受到了 C++智能指针的很大启发
- scoped_ptr 和 auto_ptr 已过时，推荐迭代到 shared_ptr 和 uniqued_ptr

5. C++其他特性

5.1. 引用参数

所有按引用传递的参数必须加上 const.

说明:在 C 语言中, 如果函数需要修改变量的值,参数必须为指针, 如 int foo(int *pval). 在 C++ 中, 函数还可以声明引用参数: int foo(int &val).

优点:定义引用参数防止出现 (*pval)++ 这样丑陋的代码. 像拷贝构造函数这样的应用也是必需的. 而且更明确, 不接受 NULL 指针.

缺点:容易引起误解, 因为引用在语法上是值变量却拥有指针的语义.

5.2. 右值引用

只在定义移动构造函数与移动赋值操作时使用右值引用, 不要使用 std::forward 功能函数. 你可能会使用 std::move 来表示将值从一个对象移动而不是复制到另一个对象.

说明:右值引用是一种只能绑定到临时对象的引用的一种, 其语法与传统的引用语法相似. 例如, void f(string&& s); 声明了一个其参数是一个字符串的右值引用的函数.

优点:

- 用于定义移动构造函数 (使用类的右值引用进行构造的函数) 使得移动一个值而非拷贝之成为可能. 例如, 如果 v1 是一个 vector<string>, 则 auto v2(std::move(v1)) 将很可能不再进行大量的数据复制而只是简单地进行指针操作, 在某些情况下这将带来大幅度的性能提升.
- 右值引用使得编写通用的函数封装来转发其参数到另外一个函数成为可能, 无论其参数是否是临时对象都能正常工作.
- 右值引用能实现可移动但不可拷贝的类型, 这一特性对那些在拷贝方面没有实际需求, 但有时又需要将它们作为函数参数传递或塞入容器的类型很有用.

- 要高效率地使用某些标准库类型，例如 std::unique_ptr，std::move 是必需的.

缺点:右值引用是一个相对比较新的特性（由 C++11 引入），它尚未被广泛理解. 类似引用崩溃，移动构造函数的自动推导这样的规则都是很复杂的.

5.3. 函数重载

函数重载，最好能让读者一看调用点就胸有成竹，不用花心思猜测调用的重载函数到底是哪一种。该规则适用于构造函数。如果打算重载一个函数,方法可以改为函数名里加上参数信息。例如，用 AppendString()和 AppendInt()，而不是重载多个 Append().

说明:你可以编写一个参数类型为 const string& 的函数，然后用另一个参数类型为 const char* 的函数重载它:

优点:通过重载参数不同的同名函数，令代码更加直观. 模板化代码需要重载，同时为使用者带来便利.

缺点:如果函数单单靠不同的参数类型而重载,读者就得十分熟悉 C++ 五花八门的匹配规则，以了解匹配过程具体到底如何。另外，当派生类只重载了某个函数的部分变体，继承语义容易令人困惑。

5.4. 缺省参数

不允许使用缺省函数参数，少数情况除外，尽可能改用函数重载。

有些人偏爱缺省参数胜于函数重载。除了以下情况，必须显式提供所有参数（注：不能通过缺省参数来省略参数）

- 位于 .cc 文件里的静态函数或匿名空间函数,毕竟都只能在局部文件里调用该函数了。
- 可以在构造函数里用缺省参数，毕竟不可能取得它们的地址。
- 可以用来模拟变长数组。

优点:当您有依赖缺省参数的函数时，您也许偶尔会修改修改这些缺省参数。通过缺省参数，不用再为个别情况而特意定义一大堆函数了。与函数重载相比，缺省参数语法更为清晰，代码少，也很好地区分了「必选参数」和「可选参数」。

缺点:缺省参数会干扰函数指针，害得后者的函数签名（function signature）往往对不上所实际要调用的函数签名。即在一个现有函数添加缺省参数，就会改变它的类型，那么调用其地址的代码可能会出错，不过函数重载就没这问题了。此外，缺省参数会造成臃肿的代码，毕竟它们在每一个调用点（call site）都有重复（acgtyrant 注：我猜可能是因为调用函数的代码表面上看来省去了不少参数,但编译器在编译时还是会在每一个调用代码里统统补上所有默认实参信息，造成大量的重复）。函数重载正好相反，毕竟它们所谓的「缺省参数」只会出现在函数定义里。

5.5. 变长数组和 alloca()

不允许使用变长数组和 alloca()，推荐改用更安全的分配器（allocator），就像 std::vector 或 std::unique_ptr<T[]>.

优点:变长数组具有浑然天成的语法. 变长数组和 alloca() 也都很高效.

缺点:变长数组和 alloca()不是标准 C++的组成部分.更重要的是，它们根据数据大小动态分配堆栈内存,会引起难以发现的内存越界 例如:“在我的机器上运行的好好的，发布后却莫名其妙的挂掉了”.

5.6. 友元

允许合理的使用友元类及友元函数.

通常友元应该定义在同一文件内,避免读者跑到其它文件查找使用该私有成员的类. 例如:经常用到友元的一个地方是将 FooBuilder 声明为 Foo 的友元,以便 FooBuilder 正确构造 Foo 的内部状态,而无需将状态暴露出来.某些情况下,将一个单元测试类声明成待测类的友元会很方便.

友元扩大了（但没有打破）类的封装边界.某些情况下,相对于将类成员声明为 public，使用友元是更好的选择，尤其是如果只允许另一个类访问该类的私有成员时.当然,大多数类都只应该通过其提供的公有成员进行互操作.

5.7. 异常

- 优先不使用 C++ 异常
- 表面上使用异常利大于弊,尤其是在新项目中.但是对现有代码,引入异常会牵连到所有相关代码.如果新项目允许异常向外扩散,跟以前未使用异常的代码整合时也是个麻烦.
- 现有代码不接受异常,在现有代码中使用异常比新项目使用的代价要大. 迁移过程比较慢,也容易出错.而异常有效替代方案,如错误代码，断言等也并不会造成严重负担.
- 并不是基于哲学或道德层面反对使用异常,是在实践的基础上，项目中使用异常会为此带来不便，因此也建议不要在开源项目中使用异常.

注: 对于异常处理，显然不是几句话能说清楚的,以构造函数为例,很多 C++书籍上都提到当构造失败时只有异常可以处理,禁止使用异常,仅是为了自身方便,是基于软件管理成本上,实际使用中还是各项目组决定

优点:

- 异常允许应用高层决定如何处理在底层嵌套函数中「不可能发生」的失败（failures），不用管那些含糊且容易出错的错误代码。
- 很多现代语言都用异常。引入异常使得 C++与 Python，Java 以及其它类 C++的语言更一脉相承。
- 有些第三方 C++ 库依赖异常，禁用异常就不好用了。
- 异常是处理构造函数失败的唯一途径。虽然可以用工厂函数或 Init() 方法代替异常，但是前者要求在堆栈分配内存，后者会导致刚创建的实例处于 ”无效“ 状态。

- 在测试框架里很好用。

缺点:

- 在现有函数中添加 throw 语句时，您必须检查所有调用点。要么让所有调用点统统具备最低限度的异常安全保证，要么眼睁睁地看异常一路欢快地往上跑，最终中断掉整个程序。举例，f() 调用 g()，g() 又调用 h()，且 h 抛出的异常被 f 捕获。当心 g，否则会没妥善清理好。
- 还有更常见的，异常会彻底扰乱程序的执行流程并难以判断，函数也许会在您意料不到的地方返回。您或许会加一大堆何时何处处理异常的规定来降低风险，然而开发者的记忆负担更重了。
- 异常安全需要 RAII 和不同的编码实践. 要轻松编写出正确的异常安全代码需要大量的支持机制. 更进一步地说，为了避免读者理解整个调用表，异常安全必须隔绝从持续状态写到“提交”状态的逻辑. 这一点有利有弊 (因为你也许不得不为了隔离提交而混淆代码). 如果允许使用异常，我们就不得不时刻关注这样的弊端，即使有时它们并不值得.
- 启用异常会增加二进制文件数据，延长编译时间（或许影响小），还可能加大地址空间的压力。
- 滥用异常会变相鼓励开发者去捕捉不合时宜，或本来就已经没法恢复的「伪异常」。比如，用户的输入不符合格式要求时，也用不着抛异常。如此之类的伪异常列都列不完。

5.8. 运行时类型识别

我们禁止使用 RTTI.

- RTTI 有合理用途但容易被滥用，在使用时务必注意. 单元测试中可以使用 RTTI，但是在其他代码中请尽量避免. 尤其是在新代码中，使用 RTTI 前务必三思. 如果你的代码需要根据不同的对象类型执行不同的行为的话，请考虑用以下的两种替代方案之一查询类型:
- 虚函数可以根据子类类型的不同而执行不同代码. 这是把工作交给了对象本身去处理.
- 如果这一工作需要在对象之外完成，可以考虑使用双重分发的方案，例如使用访问者设计模式. 这就能够在对象之外进行类型判断.
- 如果程序能够保证给定的基类实例实际上都是某个派生类的实例，那么就可以自由使用 dynamic_cast. 在这种情况下，使用 dynamic_cast 也是一种替代方案.
- 基于类型的判断树是一个很强的暗示，它说明你的代码已经偏离正轨了.

说明：RTTI 允许程序员在运行时识别 C++类对象的类型.它通过使用 typeid 或者 dynamic_cast 完成.

优点:

- RTTI 的标准替代（下面将描述）需要对有问题的类层级进行修改或重构. 有时这样的修改并不是我们所想要的，甚至是不可取的，尤其是在一个已经广泛使用的或者成熟的代码中.

- RTTI 在某些单元测试中非常有用. 比如进行工厂类测试时, 用来验证一个新建对象是否为期望的动态类型. RTTI 对于管理对象和派生对象的关系也很有用.
- 在考虑多个抽象对象时 RTTI 也很好用

缺点:

- 在运行时判断类型通常意味着设计问题. 如果你需要在运行期间确定一个对象的类型, 这通常说明你需要考虑重新设计你的类.
- 随意地使用 RTTI 会使你的代码难以维护. 它使得基于类型的判断树或者 switch 语句散布在代码各处. 如果以后要进行修改, 你就必须检查它们.

注：在类层级中加入新的子类, 代码往往会崩溃. 而且一旦某个子类的属性改变, 很难找到并修改所有受影响的代码块.不要手工实现一个类似 RTTI 方案.反对 RTTI 的理由同样适用于这些方案, 比如带类型标签的类继承体系. 而且, 这些方案会掩盖你的真实意图.

5.9. 类型转换

不要使用 C 风格类型转换. 而应该使用 C++ 风格.

- 用 static_cast 替代 C 风格的值转换, 或某个类指针需要明确的向上转换为父类指针时.
- 用 const_cast 去掉 const 限定符.
- 用 reinterpret_cast 指针类型和整型或其它指针之间进行不安全的相互转换. 仅在你对所做一切了然于心时使用.

说明:C++ 采用了有别于 C 的类型转换机制, 对转换操作进行归类.

优点:C 语言的类型转换问题在于模棱两可的操作; 有时是在做强制转换 (如 (int)3.5), 有时是在做类型转换 (如 (int)"hello"). 另外, C++ 的类型转换在查找时更醒目.

缺点:恶心的语法.

注：至于 dynamic_cast 参见 5.8. 运行时类型识别.

5.10. 流

不要使用流, 除非是日志接口需要，使用 printf 之类的代替。代码一致性胜过一切，不要在代码中使用流.

说明:流用来替代 printf() 和 scanf().

优点:有了流, 在打印时不需要关心对象的类型. 不用担心格式化字符串与参数列表不匹配 (虽然在 gcc 中使用 printf 也不存在这个问题). 流的构造和析构函数会自动打开和关闭对应的文件.

缺点:流使得 pread() 等功能函数很难执行. 如果不使用 printf 风格的格式化字符串, 某些格式化操作 (尤其是常用的格式字符串 %.*s) 用流处理性能是很低的. 流不支持字符串操作符重新排序 (%1s), 而这一点对于软件国际化很有用.

注：对这一条规则存在一些争论, 这儿给出点深层次原因.回想一下唯一性原则 (Only One Way): 希望在任何时候都只使用一种确定的 I/O 类型, 使代码在所有 I/O 处保持一致.因此,不希望由用户来决定是使用流还是 printf + read/write. 相反, 我们应该决定到底用哪一种方式. 把日志作为特例是因为日志是一个非常独特的应用, 还有历史原因.

流的支持者主张流是不二之选, 但观点并不有力. 其指出的优势也是其劣势. 流最大的优势是在输出时不用关心打印对象的类型. 这是亮点但也是不足: 开发者很容易用错类型, 而编译器不会报警，使用流时很容易造成的这类错误。虽然 printf 的格式化丑陋不堪,易读性差,但流也好不到哪儿去.

至于把流封装一下, 是可以但没必要的操作,我们的目标是使语言更紧凑, 而不是添加一些新东西.

每一种方式都各有利弊,没有最好, 只有更适合.简单性原则告诫我们必须从中选择其一,最后大多数决定采用 printf + read/write.

5.11. 前置自增和自减

对简单数值 (非对象), 两种都无所谓,对迭代器和其他模板对象使用前缀形式(++i)的自增,自减运算符.

说明:对于变量在自增(++i 或 i++) 或自减 (--i 或 i--)后表达式的值又没有没用到的情况下, 需要确定到底是使用前置还是后置的自增(自减).

优点:不考虑返回值的话, 前置自增 (++i) 通常要比后置自增 (i++) 效率更高. 因为后置自增 (或自减) 需要对表达式的值 i 进行一次拷贝. 如果 i 是迭代器或其他非数值类型, 拷贝的代价是比较大的. 既然两种自增方式实现的功能一样, 为什么不总是使用前置自增呢?

缺点:在 C 开发中, 当表达式的值未被使用时, 传统的做法是使用后置自增, 特别是在 for 循环中. 有些人觉得后置自增更加易懂, 因为这很像自然语言, 主语 (i) 在谓语动词 (++) 前.

5.12. const 用法

const 变量, 数据成员,函数和参数为编译时类型检测增加了一层保障，便于尽早发现错误. 因此,我们强烈建议在任何可能的情况下使用 const，有时改用 C++11 推出的 constexpr 更好。

- 如果函数不会修改你传入的引用或指针类型参数, 该参数应声明为 const.

- 尽可能将函数声明为 const. 访问函数应该总是 const. 其他不会修改任何数据成员，未调用非 const 函数，不会返回数据成员非 const 指针或引用的函数也应该声明成 const.
- 如果数据成员在对象构造之后不再发生变化，可将其定义为 const.

注：

- 不要过度使用 const.像 const int * const * const x; 就不合适,虽然它非常精确的描述了常量 x.关注真正有帮助意义的信息，写成 const int** x 就够了.
- 关键字 mutable 可以使用,但是在多线程中是不安全的,使用时首先要考虑线程安全.

说明:在声明的变量或参数前加上关键字 const 用于指明变量值不可被篡改 (如 const int foo). 为类中的函数加上 const 限定符表明该函数不会修改类成员变量的状态 (如 class Foo { int Bar(char c) const; };).

优点:大家更容易理解如何使用变量. 编译器可以更好地进行类型检测，相应地，也能生成更好的代码. 人们对编写正确的代码更加自信，因为他们知道所调用的函数被限定了能或不能修改变量值. 即使是在无锁的多线程编程中，人们也知道什么样的函数是安全的.

缺点:const 是入侵性的: 如果你向一个函数传入 const 变量，函数原型声明中也必须对应 const 参数 (否则变量需要 const_cast 类型转换)，在调用库函数时显得尤其麻烦.

const 的位置:

- int const *foo 形式,并不比 const int* foo 更一致、可读性更好,虽然后者遵循“const 总位于其描述的对象之后”的原则.但是该原则并不适用,不过度使用可以去除大部分原本为了一致的冗余.const 放在前面也更易读,因为在自然语言中形容词(const)是在名词(int)之前.
- 提倡但不强制 const 在前，只要保持代码的一致性即可。(**注:**不要在一些地方把 const 写在类型前面，在其他地方又写在后面，确定一种写法，然后保持一致.)

5.13. constexpr 用法

- 在 C++11 里，用 constexpr 来定义真正的常量，或实现常量初始化。
- 利用 constexpr 特性实现 C++在接口上打造真正常量机制。
- 用 constexpr 来定义真常量以及支持常量的函数。避免复杂函数定义，以使其能够与 constexpr 一起使用。
- 不要利用 constexpr 来强制代码「内联」

说明:变量可以被声明成 constexpr 以表示它是真正意义上的常量,即在编译时和运行时都不变。函数或构造函数也可以被声明成 constexpr，以用来定义 constexpr 变量。

优点:如今 constexpr 就可以定义浮点式的真·常量，不用再依赖字面值了；也可以定义用户自定义类型上的常量；甚至也可以定义函数调用所返回的常量。

缺点:若过早把变量优化成 constexpr 变量，将来又要把它改为常规变量时，挺麻烦的；当前对 constexpr 函数和构造函数中允许的限制可能会导致这些定义中解决的方法模糊。

5.14. 整型

C++内建整型中,仅使用 int.如果需要不同大小的变量,可以使用 <stdint.h> 中长度精确的整型,如 int16_t.如果变量可能不小于 2^31 (2GiB), 就用 64 位变量比如 int64_t. 注意尽管值并不会超出 int 所能够表示的范围，但在计算过程中也可能会溢出。拿不准时，干脆用更大的类型。

说明:C++ 没有指定整型的大小.通常人们假定 short 是 16 位, int 是 32 位, long 是 32 位, long long 是 64 位.

优点:保持声明统一.

缺点:C++ 中整型大小因编译器和体系结构的不同而不同.

注：

- <stdint.h> 定义了 int16_t, uint32_t, int64_t 等整型, 在需要确保整型大小时可以使用它们代替 short, unsigned long long 等. 在 C 整型中, 只使用 int. 在合适的情况下, 推荐使用标准类型如 size_t 和 ptrdiff_t.
- 如果已知整数不会太大, 我们常常会使用 int, 如循环计数. 在类似的情况下使用原生类型 int. 你可以认为 int 至少为 32 位, 但不要认为它会多于 32 位. 如果需要 64 位整型, 用 int64_t 或 uint64_t.
- 对于大整数, 使用 int64_t.
- 不要使用 uint32_t 等无符号整型, 除非你是在表示一个位组而不是一个数值, 或是你需要定义二进制补码溢出. 尤其是不要为了指出数值永不会为负, 而使用无符号类型. 相反, 你应该使用断言来保护数据.
- 如果您的代码涉及容器返回的大小（size），确保其类型足以应付容器各种可能的用法。拿不准时，类型越大越好。
- 小心整型类型转换和整型提升（acgtyrant 注：integer promotions, 比如 int 与 unsigned int 运算时，前者被提升为 unsigned int 而有可能溢出），总有意想不到的后果。

关于无符号整数被禁止：有些人甚至包括一些教科书作者推荐使用无符号类型表示非负数，并试图达到自我文档化.但在 C 语言中,这一优点被由其导致的 bug 所淹没.在比较有符号变量和无符号变量时，C 的类型提升机制会致使无符号类型的行为出乎意料.

使用断言来指出变量为非负数, 而不是使用无符号型

示例：for (unsigned int i = foo.Length()-1; i >= 0; --i) ...死循环

5.15. 64 位下的可移植性

代码应该对 64 位和 32 位系统友好. 处理打印, 比较, 结构体对齐时应切记:

对于某些类型，printf() 的指示符在 32 位和 64 位系统上可移植性不是很好.C99 标准定义了一些可移植的格式化指示符.但 MSVC 7.1 并非全部支持，而且标准中也有所遗漏

类型	不要使用	使用	备注
void *（或其他指针类型）	%lx	%p	
int64_t	%qd, %lld	%"PRId64"	
uint64_t	%qu, %llu, %llx	%"PRIu64", %"PRIx64"	
size_t	%u	%"PRIuS", %"PRIxS"	C99 规定 %zu
ptrdiff_t	%d	%"PRIdS"	C99 规定 %zd

注意：sizeof(void *)!=sizeof(int).如果需要一个指针大小的整数要用 intptr_t.

- 注意结构体对齐，尤其是要持久化到磁盘上的结构体（注：持久化 - 将数据按字节流顺序保存在磁盘文件或数据库中).在 64 位系统中,任何含有 int64_t/uint64_t 成员的类/结构体,缺省都以 8 字节在结尾对齐.如果 32 位和 64 位代码要共用持久化的结构体，需要确保两种体系结构下的结构体对齐一致.大多数编译器都允许调整结构体对齐.gcc 中可使用 __attribute__((packed)).MSVC 则提供了 #pragma pack() 和 __declspec(align())
- 创建 64 位常量时使用 LL 或 ULL 作为后缀
- 如果确实需要 32 位和 64 位系统具有不同代码，可以使用 #ifdef _LP64 指令来切分 32/64 位代码.(尽量不要这么做，如果非用不可，尽量使修改局部化)

5.16. 预处理宏

使用宏时要非常谨慎，尽量以内联函数，枚举和常量代替之.因为它意味着你和编译器看到的代码不同.这可能会导致异常行为，尤其因为宏具有全局作用域.

不过在C++中,宏不像在C中那么必不可少.以往用宏展开性能关键的代码，现在可以用内联函数替代. 用宏表示常量可被 const 变量代替.用宏“缩写”长变量名可被引用代替. 禁止用宏进行条件编译（#define 防止头文件重包含是个特例).

但宏可以做一些其他技术无法实现的事情,在一些代码库(尤其是底层库中)可以看到宏的某些特性(如用#字符串化，用##连接等等).但在使用前，仔细考虑一下能不能不用宏达到同样的目的.

下面给出的用法模式可以避免使用宏带来的问题；如果使用宏，尽可能遵守:

- 不要在 .h 文件中定义宏.
- 在马上要使用时才进行 #define，使用后要立即 #undef.
- 不要只是对已经存在的宏使用#undef，选择一个不会冲突的名称；
- 不要试图使用展开后会导致 C++ 构造不稳定的宏，不然也至少要附上文档说明其行为.
- 不要用 ## 处理函数，类和变量的名字。

5.17. 0, nullptr 和 NULL

- 整数用 0, 实数用 0.0, 指针用 nullptr 或 NULL, 字符 (串) 用 '\0'.
- 整数用 0, 实数用 0.0, 这一点是毫无争议的.
- 对于指针 (地址值), 到底是用 0, NULL 还是 nullptr. C++11 项目用 nullptr; C++03 项目则用 NULL, 毕竟它看起来像指针。实际上，一些 C++ 编译器对 NULL 的定义比较特殊，可以输出有用的警告，特别是 sizeof(NULL) 就和 sizeof(0) 不一样。
- 字符 (串) 用 '\0', 不仅类型正确而且可读性好.

5.18. sizeof

尽可能用 sizeof(varname) 代替 sizeof(type).使用 sizeof(varname) 是因为当代码中变量类型改变时会自动更新. 您或许会用 sizeof(type) 处理不涉及任何变量的代码,比如处理来自外部或内部的数据格式，这时用变量就不合适了。

5.19. auto

- 用 auto 绕过烦琐的类型名，只要可读性好就继续用
- auto 只能用在局部变量禁止用在文件作用域变量，命名空间作用域变量和类数据成员
- 禁止列表初始化 auto 变量。
- auto 还可以和 C++11 特性「尾置返回类型（trailing return type）」一起用，不过后者只能用在 lambda 表达式里

说明：C++11 中，若变量被声明成 auto, 那它的类型就会被自动匹配成初始化表达式的类型。您可以用 auto 来复制初始化或绑定引用。

优点：C++ 类型名有时又长又臭，特别是涉及模板或命名空间的时候。用 auto 重构会更好

示例：sparse_hash_map<string, int>::iterator iter = m.find(val);

-> auto iter = m.find(val);

缺点：

- 类型够明显时，特别是初始化变量时，代码才会够一目了然。
- 程序员必须会区分 auto 和 const auto& 的不同之处，否则会复制错东西。
- auto 和 C++11 列表初始化的合体令人摸不着头脑：

示例：
```
auto x(3);  // 圆括号。
auto y{3};  // 大括号。
```

它们不是同一回事：x 是 int, y 则是 std::initializer_list<int>. 其它一般不可见的代理类型也有大同小异的陷阱。

注：normally-invisible proxy types,它涉及到 C++ 鲜为人知的坑：Why is vector<bool> not a STL container?

如果在接口里用 auto，比如声明头文件里的一个常量，那么只要仅仅因为程序员一时修改其值而导致类型变化的话——API 要翻天覆地了。

5.20. 列表初始化

可以用列表初始化。

- 在 C++03 里，聚合类型（aggregate types）就可以被列表初始化了，比如数组和不自带构造函数的结构体：
- C++11 中，该特性进一步被推广，任何对象类型都可以被列表初始化。
- 用户自定义类型也可以定义接收 std::initializer_list<T> 的构造函数和赋值运算符，以自动列表初始化。
- 列表初始化也适用于常规数据类型的构造，哪怕没有接收 std::initializer_list<T> 的构造函数。

5.21. Lambda 表达式

适当使用 lambda 表达式。

- 按 format 小量使用 lambda 表达式。
- 禁用默认捕获，捕获都要显式写出来。示例：[=](int x) {return x + n;}，该写成 [n](int x) {return x + n;} 这样读者方便看出 n 是被捕获的值。
- 匿名函数始终要简短，如果函数体超过了五行，那么还不如起名（注：把 lambda 表达式赋值给对象或改用函数。
- 如果可读性更好，就显式写出 lambda 的尾置返回类型就像 auto.

说明：Lambda 表达式是创建匿名函数对象的一种简易途径，常用于把函数当参数传。C++11 首次提出 Lambdas，还提供了一系列处理函数对象的工具，比如多态包装器（polymorphic wrapper） std::function.

优点：

- 传函数对象给 STL 算法，Lambdas 最简易，可读性也好。
- Lambdas，std::functions 和 std::bind 可以搭配成通用回调机制（general purpose callback mechanism）；写接收有界函数为参数的函数也很容易了。

缺点：

- Lambdas 的变量捕获略旁门左道，可能会造成悬空指针。
- Lambdas 可能会失控；层层嵌套的匿名函数难以阅读。

5.22. 模板编程

- 模板编程有时候能够实现更简洁更易用的接口，但是更多的时候却适得其反. 因此模板编程最好只用在少量的基础组件，基础数据结构上，因为模板带来的额外的维护成本会被大量的使用给分担掉
- 在使用模板编程或者其他复杂的模板技巧的时候,一定要再三考虑一下.考虑一下团队成员的平均水平是否能够读懂并且能够维护模板代码，或者一个非 c++ 程序员和一些只是在出错的时候偶尔看一下代码的人能够读懂这些错误信息或能够跟踪函数的调用流程.如果使用递归的模板实例化或类型列表，或者元函数，又或者表达式模板，或者依赖 SFINAE，或者 sizeof 的 trick 手段来检查函数是否重载,那么这说明模板用的太多了,这些模板太复杂了,不推荐使用
- 如果使用模板编程，必须考虑尽可能的把复杂度最小化，并且尽量不要让模板对外暴漏.最好只在实现里面使用模板,然后给用户暴露的接口里面并不使用模板,这样能提高接口的可读性.并且应该在这些使用模板的代码上尽可能详细的注释.注释里应该详细的包含这些代码是怎么用的,这些模板生成出来的代码大概是什么样子的.还需要额外注意在用户错误使用你的模板代码的时候需要输出更人性化的出错信息.因为这些出错信息也是你的接口的一部分，所以你的代码必须调整到这些错误信息在用户看起来应该是非常容易理解，并且用户很容易知道如何修改这些错误

说明:模板编程指的是利用 c++ 模板实例化机制是图灵完备性，可以被用来实现编译时刻的类型判断的一系列编程技巧

优点:模板编程能够实现非常灵活的类型安全的接口和极好的性能，一些常见的工具比如 Google Test，std::tuple，std::function 和 Boost.Spirit. 这些工具如果没有模板是实现不了的

缺点:

- 模板编程所使用的技巧对于使用 c++不是很熟练的人是比较晦涩，难懂的. 在复杂的地方使用模板的代码让人更不容易读懂，并且 debug 和 维护起来都很麻烦
- 模板编程经常会导致编译出错的信息非常不友好：在代码出错的时候，即使这个接口非常的简单，模板内部复杂的实现细节也会在出错信息显示. 导致这个编译出错信息看起来非常难以理解.
- 大量的使用模板编程接口会让重构工具(Visual Assist X,Refactor for C++等等)更难发挥用途. 首先模板的代码会在很多上下文里面扩展开来，所以很难确认重构对所有的这些展开的代码有用，其次有些重构工具只对已经做过模板类型替换的代码的 AST 有用. 因此重构工具对这些模板实现的原始代码并不有效，很难找出哪些需要重构.

5.23. Boost 库

为了向阅读和维护代码的人员提供更好的可读性，只允许使用 Boost 一部分经认可的特性子集. 目前允许使用以下库:

- Call Traits : boost/call_traits.hpp
- Compressed Pair : boost/compressed_pair.hpp

- <The Boost Graph Library (BGL) : boost/graph, except serialization (adj_list_serialize.hpp) and parallel/distributed algorithms and data structures(boost/graph/parallel/* and boost/graph/distributed/*)
- Property Map : boost/property_map.hpp
- The part of Iterator that deals with defining iterators: boost/iterator/iterator_adaptor.hpp, boost/iterator/iterator_facade.hpp, and boost/function_output_iterator.hpp
- The part of Polygon that deals with Voronoi diagram construction and doesn't depend on the rest of Polygon: boost/polygon/voronoi_builder.hpp, boost/polygon/voronoi_diagram.hpp, and boost/polygon/voronoi_geometry_type.hpp
- Bimap : boost/bimap
- Statistical Distributions and Functions : boost/math/distributions
- Multi-index : boost/multi_index
- Heap : boost/heap
- The flat containers from Container: boost/container/flat_map, and boost/container/flat_set

Boost 认证还在进行中，我们会不断迭代，因此该部分的名单也会改动，注意关注。

以下库可以用，但由于如今已经被 C++11 标准库取代，不再推荐：

- Pointer Container : boost/ptr_container, 改用 std::unique_ptr
- Array : boost/array.hpp, 改用 std::array

说明: Boost 库集是一个广受欢迎，经过同行鉴定，免费开源的 C++ 库集.

优点: Boost 代码质量普遍较高，可移植性好，填补了 C++ 标准库很多空白，如型别的特性，更完善的绑定器，更好的智能指针。

缺点:某些 Boost 库提倡的编程实践可读性差，比如元编程和其他高级模板技术，以及过度 “函数化” 的编程风格.

5.24. C++11

原则上使用 C++11 的库和语言扩展，但考虑到可移植性，由项目组自行确定

C++11 特性除个别情况下原则使用。但以下特性最好不要用：

- 尾置返回类型，比如用 auto foo() -> int 代替 int foo().为了兼容于现有代码的声明风格。
- 编译时合数 <ratio>，因为它涉及一个重模板的接口风格。
- <cfenv> 和 <fenv.h> 头文件，查看编译器手册确定是否支持。
- 默认 lambda 捕获。

优点：C++11 是官方标准，目前除个别语言功能特性外已被主流 C++编译器支持。它标准化了很多早先就在用的 C++扩展，简化了不少操作，大大改善了性能和安全。

缺点：

- C++11 相对于前身，非常复杂，很多开发者也不怎么熟悉。C++11 在被迫回滚版本的项目上对代码可读性以及维护代价难以预估。
- 和 Boost 库一样，C++11 有些扩展提倡实则对可读性有害的编程实践：去除冗余检查（比如类型名）以帮助读者；鼓励模板元编程等。有些扩展在功能上与原有机制冲突，容易招致困惑以及迁移代价。

6. 命名约定

最重要的一致性规则是命名管理.命名风格能让开发以及后续维护中不需要去查找类型声明的条件快速了解某个标识符代表的含义。命名规则具有一定随意性，但相比按个人喜好命名，一致性更重要，所以无论你认为它是否重要，务必要遵守项目规范。

6.1. 通用命名规则

函数命名，变量命名，文件命名要有描述性，少用缩写.

说明：尽可能使用描述性的命名，别心疼空间，毕竟相比之下让代码易于新读者理解更重要.不要用只有项目开发者能理解的缩写,也不要通过砍掉几个字母来缩写单词.

注：一些特定的广为人知的缩写是允许的，例如用 i 表示迭代变量和用 T 表示模板参数.因此在定义时，审慎选择这类广为人知的特殊标识。

模板参数的命名应当遵循对应的分类；类型模板参数应当遵循 类型命名 的规则；非类型模板应当遵循 变量命名 的规则.

6.2. 文件命名

- 文件名要全部小写，可以包含下划线（_）或连字符（-），依照项目的约定. 如果没有约定，那么 “_”更好.
- C++文件要以 .cc 结尾,头文件以 .h 结尾. 专门插入文本的文件则以 .inc 结尾，参见头文件自包含.
- 不要使用已经存在于编译器搜索系统头文件的路径下的文件名
- 通常应尽量让文件名更加明确.
- 内联函数必须放在 .cc 文件中. 如果内联函数比较短，就直接放在 .h 中.

示例：http_server_logs.h 比 logs.h 要好.定义类时文件名一般成对出现,如 foo_bar.h 和 foo_bar.cc,对应于类 FooBar.

6.3. 类型命名

类型名称的每个单词首字母均大写，不包含下划线：MyExcitingClass，MyExcitingEnum.

说明：所有类型命名 —— 类，结构体，类型定义（typedef），枚举，类型模板参数 —— 均使用相同约定，即以大写字母开始，每个单词首字母均大写，不包含下划线.

6.4. 变量命名

变量（包括函数参数）和数据成员名一律小写，单词之间用下划线连接. 类的成员变量以下划线结尾，但结构体的就不用

示例：a_local_variable，a_struct_data_member，a_class_data_member_.

普通变量命名

示例：

```
string table_name;  // 好 - 用下划线.
string tablename;   // 好 - 全小写.
string tableName;  // 差 - 混合大小写
```

类数据成员

不管是静态的还是非静态的，类数据成员都可以和普通变量一样，但要接下划线.

示例：

```
class TableInfo {
  ...
 private:
  string table_name_;  // 好 - 后加下划线.
  string tablename_;   // 好.
  static Pool<TableInfo>* pool_;  // 好.};
```

结构体变量

不管是静态的还是非静态的，结构体数据成员都可以和普通变量一样，不用像类那样接下划线：

示例：

```
struct UrlTableProperties {
  string name;
  int num_entries;
  static Pool<UrlTableProperties>* pool;};
```

6.5. 常量命名

声明为 constexpr 或 const 的变量，或在程序运行期间其值始终保持不变的，命名时以“k” 开头，大小写混合.

示例：const int kDaysInAWeek = 7;

说明：所有具有静态存储类型的变量都应当以此方式命名.对于其他存储类型的变量，如自动变量等,这条规则是可选的.如果不采用这条规则，就按照一般的变量命名规则.

6.6. 函数命名

常规函数使用大小写混合，取值和设值函数则要求与变量名匹配。例如：
set_my_exciting_member_variable().

说明：一般来说，函数名的每个单词首字母大写（即“驼峰变量名”或“帕斯卡变量名”），没有下划线. 对于首字母缩写的单词，更倾向于将它们视作一个单词进行首字母大写（例如，写作 StartRpc() 而非 StartRPC()).

同样的命名规则同时适用于类作用域与命名空间作用域的常量，因为它们是作为 API 的一部分暴露对外的，因此应当让它们看起来像是一个函数，因为在这时，它们实际上是一个对象而非函数的这一事实对外不过是一个无关紧要的实现细节.

取值和设值函数的命名与变量一致. 一般来说它们的名称与实际的成员变量对应，但并不强制要求.

6.7. 命名空间命名

- 命名空间以小写字母命名. 最高级命名空间的名字取决于项目名称. 要注意避免嵌套命名空间的名字之间和常见的顶级命名空间的名字之间发生冲突.
- 顶级命名空间的名称应当是项目名或者是该命名空间中的代码所属的团队的名字. 命名空间中的代码，应当存放于和命名空间的名字匹配的文件夹或其子文件夹中.
- 要避免嵌套的命名空间与常见的顶级命名空间发生名称冲突. 由于名称查找规则的存在，命名空间之间的冲突完全有可能导致编译失败.尤其是不要创建嵌套的 std 命名空间. 建议使用更独特的项目标识符，而非常见的极易发生冲突的名称
- 对于 internal 命名空间，要当心加入到同一 internal 命名空间的代码之间发生冲突(由于内部维护人员通常来自同一团队，因此常有可能导致冲突).在这种情况下，请使用文件名以使得内部名称独一无二

注：不使用缩写作为名称 的规则同样适用于命名空间. 命名空间中的代码极少需要涉及命名空间的名称，因此没有必要在命名空间中使用缩写.

6.8. 枚举命名

枚举的命名应当和常量或宏一致
说明：单独的枚举值应该优先采用常量的命名方式. 但宏方式的命名也可以接受.枚举名 UrlTableErrors(以及 AlternateUrlTableErrors)是类型,所以要用大小写混合的方式.

6.9. 宏命名

不要打算使用宏，如果一定要用,像这样命名: MY_MACRO_THAT_SCARES_SMALL_CHILDREN.
说明：参考预处理宏，通常不该使用. 如果不得不用,其命名像枚举命名一样全部大写，使用下划线

6.10. 命名规则的特例

如果你命名的实体与已有 C/C++ 实体相似，可参考现有命名策略.

- bigopen(): 函数名,参照 open() 的形式
- uint: typedef
- bigpos: struct 或 class, 参照 pos 的形式
- sparse_hash_map: STL 型实体; 参照 STL 命名约定
- LONGLONG_MAX:常量,如同 INT_MAX

7. 注释

注释虽然写起来很痛苦, 但对保证代码可读性至关重要.当然也要记住:注释固然很重要, 但最好的代码应当本身就是文档.有意义的类型名和变量名,要远胜过要用注释解释的含糊不清的名字.注释是给读者看的，是下一个需要理解代码的人.

7.1. 注释风格

使用 // 或 /* */, 统一就好.

说明： // 或 /* */ 都可以; 但 // 更 常用.要在如何注释及注释风格上确保统一.

7.2. 文件注释

在每一个文件开头加入版权公告.文件注释描述了该文件的内容. 如果一个文件只声明, 或实现, 或测试了一个对象, 并且这个对象已经在它的声明处进行了详细的注释, 那么就没必要再加上文件注释. 除此之外的其他文件都需要文件注释.

说明

- 法律公告和作者信息：每个文件都应该包含许可证引用.为项目选择合适的许可证版本.(比如, Apache 2.0, BSD, LGPL, GPL)如果对原始作者的文件做了重大修改,可以考虑删除原作者信息.
- 文件内容:如果一个 .h 文件声明了多个概念, 则文件注释应当对文件的内容做一个大致的说明, 同时说明各概念之间的联系. 当然一到两行的文件注释足够了,对于每个概念的详细文档应当放在各个概念中, 而不是文件注释中.不要在 .h 和 .cc 之间复制注释, 这样的注释偏离了注释的实际意义.

7.3. 类注释

每个类的定义都要附带一份注释, 描述类的功能和用法, 除非它的功能相当明显.

说明

- 类注释应当为读者理解如何使用与何时使用类提供足够的信息, 同时应当提醒读者在正确使用此类时应当考虑的因素. 如果类有任何同步前提, 请用文档说明. 如果该类的实例可被多线程访问, 要特别注意文档说明多线程环境下相关的规则和常量使用.
- 如果你想用一小段代码演示这个类的基本用法或通常用法, 放在类注释里也非常合适.
- 如果类的声明和定义分开了(例如分别放在了 .h 和 .cc 文件中), 此时, 描述类用法的注释应当和接口定义放在一起, 描述类的操作和实现的注释应当和实现放在一起.

7.4. 函数注释

函数声明处的注释描述函数功能；定义处的注释描述函数实现.

说明

函数声明

基本上每个函数声明处前都应当加上注释，描述函数的功能和用途．只有在函数的功能简单而明显时才能省略这些注释．注释只是为了描述函数，通常，注释不会描述函数如何工作．那是函数定义部分的事情.

函数声明处注释的内容:

- 函数的输入输出.
- 对类成员函数而言: 函数调用期间对象是否需要保持引用参数, 是否会释放这些参数.
- 函数是否分配了必须由调用者释放的空间.
- 参数是否可以为空指针.
- 是否存在函数使用上的性能隐患.
- 如果函数是可重入的, 其同步前提是什么?

注释函数重载时，注释的重点应该是函数中被重载的部分，而不是简单的重复被重载的函数的注释．多数情况下，函数重载不需要额外的文档，因此也没有必要加上注释.

注释构造/析构函数时，切记读代码的人知道构造/析构函数的功能，例如“销毁这一对象”这样的注释是没有意义的．应当注明的是构造函数对参数做了什么以及析构函数清理了什么. 如果都是些无关紧要的内容, 直接省掉注释. 析构函数前没有注释是很正常的.

函数定义

如果函数的实现过程中用到了很巧妙的方式，那么在函数定义处应当加上解释性的注释．例如，所使用的编程技巧，实现的大致步骤，或解释实现理由.

不要 从 .h 文件或其他地方的函数声明处直接复制注释．简要重述函数功能是可以的，但注释重点要放在如何实现上.

7.5. 变量注释

通常变量名本身足以很好说明变量用途．某些情况下，也需要额外的注释说明.

说明

类数据成员

每个类数据成员（也叫实例变量或成员变量）都应该用注释说明用途. 如果有非变量的参数(例如特殊值，数据成员之间的关系，生命周期等)不能够用类型与变量名明确表达，则应当加上注释. 然而，如果变量类型与变量名已经足以描述一个变量，那么就不再需要加上注释.如果变量可以接受 NULL 或 -1 等警戒值，须加以说明.

全局变量

和数据成员一样，所有全局变量也要注释说明含义及用途，以及作为全局变量的原因.

7.6. 实现注释

对于代码中巧妙，晦涩，有趣，重要的地方加以注释.

说明

代码前注释

巧妙或复杂的代码段前要加注释.

行注释

比较隐晦的地方要在行尾加入注释. 在行尾空两格进行注释. 如果你需要连续进行多行注释，可以使之对齐获得更好的可读性

函数参数注释

- 如果函数参数的意义不明显，考虑用下面的方式进行弥补:
- 如果参数是一个字面常量，并且这一常量在多处函数调用中被使用，用以推断它们一致，你应当用一个常量名让这一约定变得更明显，并且保证这一约定不会被打破.
- 考虑更改函数的签名，让某个 bool 类型的参数变为 enum 类型，这样可以让这个参数的值表达其意义.
- 如果某个函数有多个配置选项，你可以考虑定义一个类或结构体以保存所有的选项，并传入类或结构体的实例. 这样的方法有许多优点，例如这样的选项可以在调用处用变量名引用，这样就能清晰地表明其意义. 同时也减少了函数参数的数量，使得函数调用更易读也易写. 除此之外，以这样的方式，如果你使用其他的选项，就无需对调用点进行更改.
- 用具名变量代替大段而复杂的嵌套表达式.
- 万不得已时，才考虑在调用点用注释阐明参数的意义.

禁止行为：

- 不要描述显而易见的现象，
- 永远不要用自然语言翻译代码作为注释,除非代码难度很高，对深入理解 C++的读者来说代码的行为都是不明显的.
- 要假设读代码的人 C++水平比你高,即便他/她可能不知道你的用意
- 所提供的注释应当解释代码为什么要这么做和目的,或者最好让代码自文档化.

7.7. 标点, 拼写和语法

注意标点, 拼写和语法; 写的好的注释比差的要易读的多.

说明:注释的通常写法是包含正确语法和结尾句号的完整叙述性语句.多数情况下,完整的句子比片段可读性更高.短注释,比如代码行尾注释,可以随意点,但依然要注意风格的一致性.

7.8. TODO 注释

对那些临时的, 短期的解决方案, 或已经够好但仍不完美的代码使用 TODO 注释.

说明: TODO 注释要使用全大写的字符串 TODO,在随后的圆括号里写上名字, 邮箱, bug ID, 或其它身份标识和与这一 TODO 相关的 issue.目的是让添加注释的人可根据规范的 TODO 格式进行查找.添加 TODO 注释并不意味着要自己修正,因此当你加上带有姓名的 TODO 时, 一般都是写上自己的名字, 方便告知更多细节给维护人员。

如果加 TODO 是为了“将来某一天做某事”,可以附上一个明确的时间或一个明确的事项

7.9. 弃用注释

通过弃用注释（DEPRECATED comments）标记某接口点已弃用.

可以写上包含全大写的 DEPRECATED 的注释, 以标记某接口为弃用状态. 注释可以放在接口声明前, 或者同一行.在 DEPRECATED 一词后, 在括号中留下您的名字, 邮箱以及其他身份标识.

弃用注释应当包涵简短而清晰的指引, 以帮助其他人修复其调用点.在 C++中,可以将一个弃用函数改造成一个内联函数, 这一函数将调用新的接口.

标记接口为 DEPRECATED 并不会真正完成弃用, 还需要修正调用点,修正好的代码一般不会再涉及弃用接口点,改用新接口点. 在完成修复后, 代码可以做清除工作了。

8. 格式

每个人都可能有自己的代码风格和格式, 但如果一个项目中的所有人都遵循同一风格的话, 这个项目就能更顺利地进行. 每个人未必能同意下述的每一处格式规则, 而且其中的不少规则需要一定时间的适应, 但整个项目服从统一的编程风格是很重要的, 只有这样才能让所有人轻松地阅读和理解代码.

8.1. 行长度

每一行代码字符数不超过 80.这条规则是有争议的, 但很多已有代码都遵照这一规则。

注:

- 如果无法在不伤害易读性的条件下进行断行，那么注释行可以超过 80 个字符，这样可以方便复制粘贴.例如,带有命令示例或 URL 的行可以超过 80 个字符.
- 包含长路径的 #include 语句可以超出 80 列.
- 头文件保护 可以无视该原则.

优点：很多人同时并排开几个代码窗口，根本没有多余的空间拉伸窗口. 大家都把窗口最大尺寸加以限定，并且 80 列宽是传统标准.

缺点：反对该原则的人则认为更宽的代码行更易阅读. 80 列的限制是上个世纪 60 年代的大型机的古板缺陷；现代设备具有更宽的显示屏，可以很轻松地显示更多代码.

8.2. 非 ASCII 字符

尽量不使用非 ASCII 字符，使用时必须使用 UTF-8 编码.

说明：不应将用户界面的文本硬编码到源代码中,非 ASCII 字符应当很少被用到.特殊情况下可以适当包含此类字符.例如,代码分析外部数据文件时，可以适当硬编码数据文件中作为分隔符的非 ASCII 字符串或单元测试代码可能包含非 ASCII 字符串.此情况下,应使用 UTF-8 编码，因为很多工具都可以理解和处理 UTF-8 编码.

十六进制编码也可以，能增强可读性的情况下推荐——比如"\xEF\xBB\xBF"或更简洁地写作 u8"\uFEFF",在 Unicode 中是零宽度无间断的间隔符号,如果不用十六进制直接放在 UTF-8 格式的源文件中，是看不到的.

注："\xEF\xBB\xBF" 通常用作 UTF-8 with BOM 编码标记

使用 u8 前缀把带 uXXXX 转义序列的字符串字面值编码成 UTF-8. 不要用在本身就带 UTF-8 字符的字符串字面值上，因为如果编译器不把源代码识别成 UTF-8，输出就会出错.

别用 C++11 的 char16_t 和 char32_t，它们和 UTF-8 文本没有关系，wchar_t 同理，除非你写的代码要调用 Windows API，后者广泛使用了 wchar_t.

8.3. 空格还是制表位

只使用空格，每次缩进 4 个空格.

说明：我们使用空格缩进.不要在代码中使用制表符.

注：不是连按空格，是 tab，但是转换出来就是空格了

8.4. 函数声明与定义

返回类型和函数名在同一行，参数也尽量放在同一行，如果放不下就对形参分行，分行方式与 函数调用 一致.

注意：

- 使用好的参数名.
- 只有在参数未被使用或者其用途非常明显时，才能省略参数名.
- 如果返回类型和函数名在一行放不下，分行.
- 如果返回类型与函数声明或定义分行了，不要缩进.
- 左圆括号总是和函数名在同一行.
- 函数名和左圆括号间永远没有空格.
- 圆括号与参数间没有空格.
- 左大括号另起新行，函数短且嵌套层次低时在最后一个参数同一行的末尾处
- 右大括号总是单独位于函数最后一行，或者与左大括号同一行.
- 右圆括号和左大括号间总是有一个空格.
- 所有形参应尽可能对齐.
- 缺省缩进为 2 个空格.
- 换行后的参数保持 4 个空格的缩进.
- 未被使用的参数如果其用途不明显的话，在函数定义处将参数名注释起来
- 属性，和展开为属性的宏，写在函数声明或定义的最前面，即返回类型之前:

8.5. Lambda 表达式

Lambda 表达式对形参和函数体的格式化和其他函数一致;捕获列表同理,表项用逗号隔开.

说明：若用引用捕获，在变量名和 & 之间不留空格，短 lambda 就写得和内联函数一样.

8.6. 函数调用

- 要么一行写完函数调用,没有其它顾虑的话,尽可能精简行数,把多个参数适当地放在同一行里.
- 如果同一行放不下，可断为多行，后面每一行都和第一个实参对齐，左圆括号后和右圆括号前不要留空格：
- 参数也可以放在次行，缩进四格
- 把多个参数放在同一行以减少函数调用所需的行数,除非影响到可读性.我不赞同每个参数都独立成行更好读且方便编辑参数的说法.比起所谓的参数编辑,我更看重可读性,且后者比较好办：
- 如果一些参数本身就是略复杂的表达式，且降低了可读性，那么可以直接创建临时变量描述该表达式，并传递给函数：
- 如果某参数独立成行，对可读性更有帮助的话，那也可以如此做. 参数的格式处理应当以可读性而非其他作为最重要的原则.
- 此外，如果一系列参数本身就有一定的结构，可以酌情地按其结构来决定参数格式

8.7. 列表初始化格式

平时怎么格式化函数调用，就怎么格式化 列表初始化.

说明：如果列表初始化伴随着名字，比如类型或变量名，格式化时将将名字视作函数调用名，{} 视作函数调用的括号. 如果没有名字，就视作名字长度为零.

8.8. 条件语句

倾向于不在圆括号内使用空格. 关键字 if 和 else 另起一行.

说明

- 对基本条件语句有两种可以接受的格式.在圆括号和条件之间有空格,或者没有.
- 最常见的是没有空格的格式.其实都可以, 重要的是保持一致. 在修改一个文件时, 要参考已有格式. 如果写新代码,参考目录下或项目中其它文件.
- 如果能增强可读性, 简短的条件语句允许写在同一行. 只有当语句简单并且没有使用 else 子句时使用。如果语句有 else 分支则不允许
- 通常单行语句不需要使用大括号, 用也没问题; 复杂的条件或循环语句用大括号可读性会更好. 也有一些项目要求 if 必须总是使用大括号:
- 但如果语句中某个 if-else 分支使用了大括号的话, 其它分支也必须使用:

注：所有情况下 if 和左圆括号间都有个空格. 右圆括号和左大括号之间也要有个空格:

8.9. 循环和开关选择语句

switch 语句可以使用大括号分段, 以表明 cases 之间不是连在一起的. 在单语句循环里, 括号可用可不用. 空循环体应使用 {} 或 continue.

说明

switch 语句中的 case 块使用大括号的话, 要按照下文所述的方法.

- 如果有不满足 case 条件的枚举值,switch 应该总是包含一个 default 匹配 (如果有输入值没有 case 去处理,编译器将给出 warning). 如果 default 应该永远执行不到, 简单的加条 assert:
- 在单语句循环里, 括号可用可不用：空循环体应使用 {} 或 continue, 而不是一个简单的分号.

8.10. 指针和引用表达式

句点或箭头前后不要有空格. 指针/地址操作符 (*, &) 之后不能有空格.

注:声明指针变量或参数时, 星号与类型或变量名紧挨都可以:在单个文件内要保持风格一致, 修改现有文件时, 要遵照文件风格.

8.11. 布尔表达式

如果一个布尔表达式超过标准行宽, 断行方式要统一.此外,直接用符号形式的操作符,比如 && 和 ~,不要用词语形式的 and 和 compl.

8.12. 函数返回值

不要在 return 表达式里加上非必须的圆括号.

说明：只有在写 x = expr 要加上括号的时候才在 return expr; 里使用括号.

8.13. 变量及数组初始化

用 =, () 和 {} 均可.

说明：务必小心列表初始化 {...} 用 std::initializer_list 构造函数初始化出的类型. 非空列表初始化就会优先调用 std::initializer_list, 不过空列表初始化除外, 后者原则上会调用默认构造函数. 为了强制禁用 std::initializer_list 构造函数, 请改用括号. 列表初始化不允许整型类型的四舍五入, 这可以用来避免一些类型上的编程失误.

8.14. 预处理指令

预处理指令不要缩进, 从行首开始.

说明：即使预处理指令位于缩进代码块中, 指令也应从行首开始.

8.15. 类格式

访问控制块的声明依次序是 public:, protected:, private:, 每个都缩进 1 个空格.

注意:

- 所有基类名应在 80 列限制下尽量与子类名放在同一行.
- 关键词 public:, protected:, private: 要缩进 1 个空格.
- 除第一个关键词 (一般是 public) 外, 其他关键词前要空一行. 如果类比较小的话也可以不空.
- 这些关键词后不要保留空行.
- public 放在最前面, 然后是 protected, 最后是 private.
- 关于声明顺序的规则请参考 声明顺序 一节.

8.16. 构造函数初始值列表

构造函数初始化列表放在同一行或按四格缩进并排多行.

8.17. 命名空间格式化

命名空间内容不缩进.

说明：命名空间 不要增加额外的缩进层次, 不要在命名空间内缩进:声明嵌套命名空间时, 每个命名空间都独立成行.

8.19. 水平留白

水平留白的使用根据在代码中的位置决定，不要在行尾添加没意义的留白.

添加冗余留白会给其他人编辑造成额外负担. 行尾不要留空格,如果确定一行代码已经修改完毕，将多余的空格去掉，或者在专门清理空格时去掉

注:现在大部分代码编辑器稍加设置后，都支持自动删除行首/行尾空格，如果不支持，考虑换一款编辑器或 IDE

8.19. 垂直留白

垂直留白越少越好.

说明:这已经是原则问题了: 不在万不得已，不要使用空行. 尤其是: 两个函数定义之间的空行不要超过 2 行，函数体首尾不要留空行，函数体中也不要随意添加空行.

基本原则是同一屏可以显示的代码越多,越容易理解程序的控制流.当然，过于密集的代码块和过于疏松的代码块同样难看，这取决于你的判断. 但通常是垂直留白越少越好.

下面的规则可以让加入的空行更有效:

- 函数体内开头或结尾的空行可读性微乎其微.
- 在多重 if-else 块里加空行或许有点可读性.

9. 规则特例

前面说明的编程习惯基本都是强制性的. 但所有优秀的规则都允许例外，这里就是探讨这些特例.

9.1. 现有不合规范的代码

对于现有不符合既定编程风格的代码可以网开一面.

说明：当你修改使用其他风格的代码时,为了与代码原有风格保持一致可以不使用本规范约定.如果不放心,可以与代码原作者或现在的负责人员商讨.记住：一致性也包括原有的一致性.

9.2. Windows 代码

Windows 程序员有自己的编程习惯，主要源于 Windows 头文件和其它 Microsoft 代码. 我希望任何人都可以顺利读懂代码，所以针对 Windows 平台的 C++ 编程给出一个单独的指南.

说明：如果你习惯使用 Windows 编码风格，有必要重申一下某些规则:

- 不要使用匈牙利命名法 (比如把整型变量命名成 iNum).

- Windows 定义了很多原生类型的同义词 (注:这一点,我也很反感),如 DWORD, HANDLE 等.在调用 Windows API 时这是完全可以接受甚至鼓励的. 但即使如此, 还是尽量使用原有的 C++类型, 例如使用 const TCHAR *而不是 LPCTSTR.
- 使用 Microsoft Visual C++进行编译时,将警告级别设置为 3 或更高,并将所有警告(warnings)当作错误(errors)处理.
- 不要使用 #pragma once; 应该使用 Google 的头文件保护规则（注：#define 保护）. 头文件保护的路径应该相对于项目根目录
- 除非万不得已, 不要使用任何非标准的扩展, 如#pragma 和__declspec. 使用 __declspec(dllimport) 和 __declspec(dllexport) 是允许的, 但必须通过宏, 比如 DLLIMPORT 和 DLLEXPORT,这样其他人在分享使用这些代码时可以轻松禁用这些扩展.

但在 Windows 上仍然有一些需要违反的规则:

- 通常禁止使用多重继承,但在使用 COM 和 ATL/WTL 类时可以使用.为了实现 COM 或 ATL/WTL 类/接口, 你可能不得不使用多重实现继承.
- 虽然代码中不应该使用异常, 但是在 ATL 和部分 STL（包括 Visual C++ 的 STL) 中异常被广泛使用.使用 ATL 时,应定义 _ATL_NO_EXCEPTIONS 以禁用异常. 你需要试验下是否能够禁用 STL 的异常, 如果无法禁用,可以启用编译器异常. (注意这只是为了编译 STL, 自己的代码里仍然不应当包含异常处理).
- 通常为了利用头文件预编译,每个每个源文件的开头都会包含一个名为 StdAfx.h 或 precompile.h 的文件.为了使代码方便与其他项目共享, 避免显式包含此文件 (除了在 precompile.cc 中), 使用/FI 编译器选项以自动包含该文件.
- 资源头文件通常命名为 resource.h 且只包含宏, 这一文件不需要遵守本风格指南.

（三）Python

Python 会用，会调库就行，目前基本涉及不到程序设计，下次迭代再加

——陈旭

四．Doxygen 格式注释规范

该部分规范不作强制要求，仅作为推荐。

Doxygen 是一种开源跨平台的,以类似 JavaDoc 风格描述的文档系统,完全支持 C、C++、Java、Objective-C 和 IDL 语言，部分支持 PHP、C#。注释的语法与 Qt-Doc、KDoc 和 JavaDoc 兼容。Doxgen 可以从一套归档源文件开始，生成 HTML 格式的在线类浏览器，或离线的 LATEX、RTF 参考手册。

1. 常用指令说明

命令	生成字段名	说明
@{	模块开始	
@}	模块结束	
@addtogroup	添加到一个组	
@attention	注意	
@author	作者	
@brief	简要注释	
@bug	缺陷	链接到所有缺陷汇总的缺陷列表
@class	类名	
@code	引用代码段	代码块开始，与"nendcode"成对使用
@date	日期	
@defgroup	模块名	
@details	详细注释	
@endcode	引用代码段结束	代码块结束，与"ncode"成对使用
@exception	函数抛异常描述	
@file	文件名	可以默认为空，DoxyGen会自己加
@fn	函数说明	
@include	包含文件	
@ingroup	加入到一个组	
@name	分组名	
@note	开始一个段落	用来描述一些注意事项
@par	开始一个段落	段落名称描述由你自己指定
@param	函数参数	用在函数注释中
@post	函数后置条件	比如对系统状态的影响或返回参数的结果预期
@pre	函数前置条件	比如对输入参数的要求
@remarks	备注	
@return	函数返回值描述	用在函数注释中
@retval	描述返回值意义	
@see	see	also字段
@since	从哪个版本后开始有这个函数的	
@test	被标记的代码会在Test列表中出现	
@todo	TODO	链接到所有TODO汇总的TODO列表
@version	版本号	
@warning	函数使用中需要注意的地方	

示例：

```cpp
/*********************************************************************************
 * Copyright(c) 2000-3000 Company Name                                           *
 * All rights reserved.                                                          *
 *                                                                               *
 * @license                                                                      *
 *                                                                               *
 *********************************************************************************/
/**
 * @file     文件名
 * @brief    简要注释
 * @details  详细注释
 * @version  版本
 * @author   作者
 * @email    邮箱
 * @date     日期
 *
 *
 * Change History:
 * @author
 * @email
 * @date
 * @brief
 *
 */
#pragma once

//---------------------------------------------------------------------
//  简要注释
//---------------------------------------------------------------------
#define PI 3.14
#define MAX 10000
#define MIN 0
#define ID 1024

/**
 * @brief    简要注释
 * @details  详细注释
 */
namespace doxygentest
{
    /**
     * @class    类名
     * @brief    简要注释
     * @details  详细注释
     * @author   作者
     * @note 用来描述一些注意事项
     *
     */
    class CDoxygenTest
    {
    public:
        CDoxygenTest();
        ~CDoxygenTest();
    private:
        //简要说明(单行注释)
        int m_num;  ///< 变量注释
    public:
        /**
         * @brief        简要注释
         * @details      详细注释
         *
         * @param[in]    parameter_1 （输入参数）
         * @param[out]   parameter_2 （输出参数）
         * @return       函数返回值描述
         * @retval       描述返回值意义  true
         * @retval       描述返回值意义  false
         *
         * @note         用来描述一些注意事项
         * @remarks      备注
         */
        bool SetNum(int intNum);
    };

    /**
     * @struct   结构名
     * @brief    简要注释
     * @details  详细注释
     * @author   作者
     * @note 用来描述一些注意事项
     *
     */
    struct MyStruct
    {
        int intVar1;    ///< 变量注释
        int intVar2;    ///< 变量注释
    };

    /**
     * @enum     枚举名
     * @brief    简要注释
     * @details  详细注释
     * @author   作者
     * @note 用来描述一些注意事项
     *
     */
    enum MyEnum
    {
        red,    ///< 红色
        green,  ///< 绿色
        blue    ///< 蓝色
    };
}
```

单行注释

需要注释代码的上一行，以//开头+注释类容

```
//简要说明(单行注释)
int m_num;    ///< 变量注释
```

标注总叙

```
//--------------------------------------------------------------
//   简要注释
//--------------------------------------------------------------
```

头文件注释

```
/*****************************************************************************
* Copyright(c) 2000-3000 Company Name                                        *
* All rights reserved.                                                       *
*                                                                            *
* @license                                                                   *
*                                                                            *
*****************************************************************************/
/**
* @file       文件名
* @brief      简要注释
* @details    详细注释
* @version    版本
* @author     作者
* @email      邮箱
* @date       日期
*
*
* Change History:
* @author
* @email
* @date
* @brief
*
*/
```

命名空间

```
/**
* @brief 简要注释
* @details    详细注释
```

```
*/
```

类、结构、枚举注释

类

```
/**
* @class 类名
* @brief 简要注释
* @details    详细注释
* @author     作者
* @note       用来描述一些注意事项
*
*/
```

结构

```
/**
* @struct     结构名
* @brief 简要注释
* @details    详细注释
* @author     作者
* @note  用来描述一些注意事项
*
*/
```

枚举

```
/**
* @struct     结构名
* @brief 简要注释
* @details    详细注释
* @author     作者
* @note  用来描述一些注意事项
*
*/
```

函数注释

```
/**
* @brief      简要注释
* @details        详细注释
*
* @param[in] parameter_1   （输入参数）
* @param[out]parameter_2   （输出参数）
```

```
* @return          函数返回值描述
* @retval          描述返回值意义    true
* @retval          描述返回值意义    false
*
* @note            用来描述一些注意事项
* @remarks         备注
*/
```

变量标注

变量名所在行后面+ ///< 变量注释