**Google C++ Style Guide**

## General

原则1：Naming，Comments，Formatting小节通常不影响代码功能及性能，而且每个人都会用到，为了保证代码统一性，必须遵循。

原则2：Header Files，Scoping，Classes涉及到对c/c++语法特性的正确优化使用，遵循它能够提高代码的正确性，高性能及可读性等，用的时候应当遵循。

原则3：与现有代码规范有冲突时，以保持统一为准。

## Header Files

通常每个.cc文件都应该对应有一个.h文件，除非是单元测试或者仅仅包含main函数的比较小的cc文件。

正确的使用.h文件，能够大幅改善代码可读性，尺寸和性能。

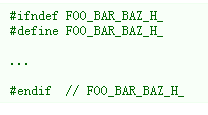
### The #define Guard

**#define 保护**

所有的头文件都应该使用#define防止重复包含。

命名格式应该使用*<PROJECT>*\_*<PATH>*\_*<FILE>*\_H\_，为了保证唯一性，命名应该基于文件所在项目源码树的全路径。

例如，foo/src/bar/baz.h文件按如下方式保护：



### Forward Declarations

**前置声明**

当在使用头文件中的函数定义时，总是使用#include包含这个头文件

当使用类模板，倾向使用#include包含类的文件

当使用普通类，使用前置声明是可以的，但是要注意前置声明不满足或者不合适的情况，如果确定不了时，使用#include包含合适的头文件

不要使用指针替换数据成员变量仅仅是为了避免使用#include

总是#include实际提供你需要的声明/定义的头文件，不要依赖间接引用的头文件传递进来的符号。

### Inline Functions

**内联函数**

只有取值和设值函数，以及其它短的，性能关键性的函数才内联。

不要内联一个超过10行的函数。

不要内联一个包含循环或者switch语句的函数，除非通常情况下，循环和switch语句从来不会执行。

不要内联虚函数和递归函数。

### Function Parameter Ordering

**函数参数顺序**：

当定义一个函数时，参数顺序是：输入，然后是输出。

排列函数参数时，将仅做输入的参数放到所有输出参数的前面。

不要因为是新增加的参数，把它们放到参数表的最后，要把新的只做输入的参数放到所有输出参数的前面。

这不是一个硬性的规则。和已有代码保持统一更重要。

### Names and Order of Includes

**命名和包含顺序**

标准的顺序：C库，C++库，其它库的.h文件，本项目的.h文件

项目中所有头文件应该按照项目源码树排列，避免使用Unix快捷目录。如当前目录（.）和父目录（..）。例如，google-awesome-project/src/base/logging.h应该这样被包含：



dir/foo.cc和dir/foo\_test.cc文件主要目的是实现或者测试dir2/foo2.h文件，头文件包含顺序应该：

1. dir2/foo2.h （优先包含）

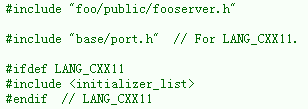
2. C库文件

3. C++库文件

4. 其他库文件

5. 本项目头文件

异常情况：有时候系统相关的代码需要条件包含，可以将条件包含代码放到其他包含之后。当然，保证你的系统相关代码短小而且小范围的。例如：



## Scoping

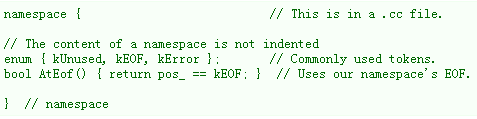
**作用域**

### Namespaces

**命名空间**

**未命名命名空间**

未命名命名空间在.cc文件中是允许的，而且提倡这么做来避免运行时命名冲突：

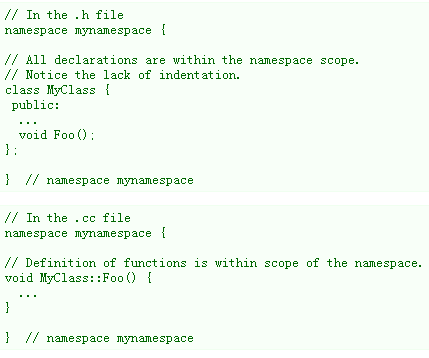


但是，文件作用域内，一些与某个类有关的声明也可以声明为那个类的类型、静态数据成员或是静态函数成员，而不是声明在未命名空间内。

不能在.h文件中使用未命名空间。

**命名空间**

命名空间将源文件中除了头文件包含，全局声明/定义，和其他命名空间中的前置声明以外的内容全部包含起来：



不要声明命名空间std下的任何内容，即使是标准库中的前置声明。声明命名空间std下的实体会导致未定义的行为，如不可移植。声明来自标准库中的实体，就直接包含合适的头文件。

不要使用using指示符来引入命名空间中所有可用的名字。



你可以在.cc文件中的任何地方使用using指示符，也可以在.h文件中的函数，方法或者类中使用（即不能在公共作用域内使用）。



允许在.cc文件的任何地方为命名空间起别名，在.h文件的命名空间内的任何地方，函数和方法内也可以这么做。



公共的头文件（在项目之外也可见）和被它们包含的头文件应该避免定义别名。

不要使用内联命名空间。

### Nonmember, Static Member, and Global Functions

**非成员函数，静态成员函数和全局函数**

非成员函数应该不依赖于外部变量，且应该在命名空间内。

如果为了将静态成员函数组合起来而创建一个没有共享静态数据的类，不如将它们放到命名空间中。

定义在同一个编译单元内的函数在被其它编译单元调用时可能会引入不必要的耦合和链接时依赖；静态成员函数尤其容易受此影响。此时可考虑提取出一个新的类或是将这些函数放到独立库的命名空间内。

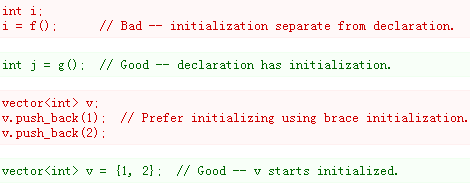
如果你必须定义一个非成员函数，而且仅仅在.cc文件中需要，使用未命名空间或者使用static限制它的作用域。

### Local Variables

**局部变量**

将一个函数的变量限制最小作用域内，并且在声明处进行初始化。

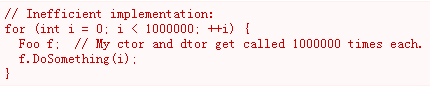
尽可能在本地作用域中声明变量，并且尽可能在第一次使用的地方。这样更方法读者找到声明，查看变量类型和初始值。尤其是初始化应该用来取代声明和赋值。例如：



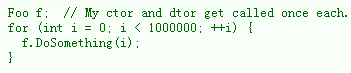
GCC能正确实现for (int i = 0; i < 10; ++i)（i的作用范围仅限于循环体内），所以你可以在同一个作用域中重复使用i。在if和while语句中GCC也能正确限制变量的作用域。例如：



如果变量是一个类的对象，每次进入这个作用域都会调用它的构建函数并创建对象，离开作用域时同样会调用析构函数。



更高效的方法是在循环外面声明这个变量：



### Static and Global Variables

**静态变量和全局变量**

禁止使用类类型的静态变量或全局变量.

只允许POD类型的静态变量。只有内置类型int，char，foloat或指针，或是数组和结构体才是POD类型。

不允许静态POD变量利用函数的返回值进行初始化，除非那个函数（如getenv()或getpid()）不依赖于其它的全局符号。

如果你需要一个类类型的静态或全局变量，考虑在main函数或是在pthread\_once函数初始化一个指针（永远不会被释放）。要注意它必须是原始的指针而不是智能指针，因为智能指针的析构器同样存在我们极力避免的析构顺序的问题。

## Classes

类是C++最基本的代码单元。显然它们被广泛使用。本节列出了你写一个类时，应遵循的主要原则。

### Doing Work in Constructors

**构造函数中做的工作**

要避免在构造函数内进行复杂的初始化（尤其是可能会失败的初始化或是调用虚函数）。

构造函数不应该调用虚函数或是进行可能引起非致命的错误的操作。如果你的对象需要有意义的初始化，考虑用一个工厂函数或是Init()方法。

### Initialization

**初始化**

使用类内部成员初始化器来做简单的初始化，特别是当成员变量必须在超过一个构造函数里面使用相同的方式来初始化时。

如果你的类定义成员变量没有在类内部初始化，而且它没有其它的构造函数，你必须定义一个缺省的构造函数（不带任何参数）。将对象初始化为统一和有效的内部状态。

如果你的类从已经存在的类派生，但是你没有新增成员变量，你不需要定义缺省的构造函数。

### Explicit Constructors

**显式构造函数**

我们要求所有的单个参数构造函数使用explicit关键字修饰。总是将explicit放在单个参数构造函数的前面：explicit Foo(string name);

一个例外是拷贝构造函数，这是我们少有的允许隐式转换，不需要使用explicit。还有对其他类透明封装的类也可以例外。这些例外应该被清楚的注释。

最后，仅仅有初始化列表的构造函数可以是非explicit。这允许你使用这样的初始化列表构造你自己的类型，例如MyType m = {1, 2}。

### Copy Constructors

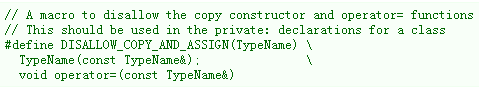
**拷贝构造函数**

只有少数类需要被复制。大多数类即不需要拷贝构造函数也不需要赋值操作函数。

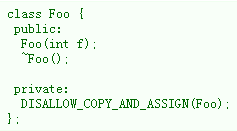
如果你的类需要被复制，优先提供拷贝函数，比如CopyFrom()或者Clone()，而不是拷贝构造函数，如果实在需要，同时提供拷贝构造函数和赋值操作符。

如果你的类不需要拷贝构造函数和赋值操作符，你必须显式的禁止它。为了这么做，可以在private:中增加假的拷贝函数和赋值操作符声明，但是不提供相应的实现(这样任何对它们的使用会导致链接错误)。

为了方便，可以定义成宏DISALLOW\_COPY\_AND\_ASSIGN：



然后在类中这样使用：



### Structs vs. Classes

**结构体VS类**

struct和class关键字在C++中的行为几乎是相同的。我们在这两个关键字上添加自己的语义理解，这样为自己定义的数据类型加上合适的关键字。

struct应该用来定义仅仅承载数据的对象，可能包含常量，但是不能有除了存取数据成员以外的其他功能。通过直接访问数据成员来存取成员，而不是通过函数调用。除了初始化数据成员，不能提供其他行为，例如构造函数，析构函数，Initialize(), Reset(), Validate()。

如果要求更多功能，使用类更合适。如果有疑虑，使用类。

注意结构体和类中的成员变量具有不同的命名规则。

### Inheritance

所有的继承应该都是公有的。如果你想提供私有继承，你应该包含一个基类的实例作为成员变量来代替。

不要过度的使用实现继承。组合通常更合适。尽量严格的使用“is-a”关系的继承。如果说Bar是一种 Foo是一种合理的说法，Bar可以从Foo继承。

如果你的类有虚拟函数，析构函数应该是虚拟的。

使用protected关键字来限制可能被子类访问的成员函数。

成员变量应该是私有的。

当重定义一个继承的虚拟函数时，在派生类中显式的使用virtual关键字定义。原因：如果virtual关键字被漏掉，读者不得不去检查这个类的所有祖先来确定这个函数是否是虚拟的。

### Multiple Inheritance

**多重继承**

多重继承只有在所有的基类都是纯接口时才允许使用，其中第一个基类可以例外。为了能明确它们纯接口，它们的名字必须带着Interface的后缀。

### Interfaces

**接口**

命名以Interface结尾。

它只有纯虚函数和静态方法（但是参考下面的析构函数说明）。

它没有非静态的数据成员。

它不需要任何构造函数定义。如果提供了构造函数，必须没有参数而且是被保护的。

如果它是一个子类，它同样只能从满足这些条件且有着Interface后缀标记的类中派生。

一个接口类不能被实例化，因为其中定义了纯虚函数。为了确保所有的接口实现能被正确的销毁，这个接口必须声明一个虚拟析构函数（这里第一条规则例外，这应该不是纯虚函数）。

### Operator Overloading

**操作符重载**

一般来说不要重载操作符

不要重载operator""，即不要引入用户定义常量。

但是也有一些少数情况，你需要重载操作符来与模板或是标准C++类连接使用（例如operator<<(ostream&, const T&)写日志）。只有被证明是完全合理的才能重载，但还是要尽量避免重载。尤其是，不要为了类能充当STL容器的键值就重载“==“或”<“；你应该在声明容器的时候使用”==“和”<“的仿函数算子。

一些STL算法要求重载operator==，你可以这么做，但需要在文档内写明为什么。

### Access Control

**访问控制**

定义数据成员为私有的，并在需要的时候提供取值函数（因为技术原因，在使用Google测试工具时，允许测试装置类的数据成员是protected的）。

典型的例子，一个变量被称foo\_，它的取值函数就是foo()。你也可能需要一个设值函数set\_foo()。例外：静态常量数据成员（static const）（名为kFoo）不需要是私有的。

取值函数经常在头文件中定义为内联的。

### Declaration Order

类的定义应该从公开成员部分开始（public:），接着是受保护成员（protected:），最后是私有成员（private:）。如果哪部分是空的就省略掉。

在每部分中，声明通常应该按下面的顺序排列：

* 自定义别名和枚举
* 常量（静态常量）
* 构造函数
* 析构函数
* 方法，包括静态方法
* 数据成员（除了静态常量成员）

友元声明应该总是放在私有部分，禁止复制和赋值宏（DISALLOW\_COPY\_AND\_ASSIGN）应该放到私有部分的最后。它应该是类中最后的东西。

在对应的.cc文件中，方法的定义的顺序应该尽量与它被声明的顺序相同。

### Write Short Functions

**尽量编写短而专一的函数**

如果一个函数超过了40行，就要考虑能否在不伤害到程序结构的前提下将它拆分。

不要害怕修改已经存在的代码：和这样的函数工作会很困难，你会发现错误很难调试，或者你想将它用在一些不同的场合，考虑把这个函数拆分成几段更小更容易管理的片段。

### Reference Arguments

**引用参数**

函数参数表中所有的引用都必须使用const修饰：



### Function Overloading

**函数重载**

仅仅在这种情况下使用重载函数（包括构造函数）：读者一看调用语句就能清楚发生了什么，而不用首先去查看究竟哪个版本的函数被调用了。

如果你想要重载一个函数，考虑一下将函数的名字改成带有一定的参数的信息而不是重载，例如，AppendString()和AppendInt()而不是只有一个Append()。

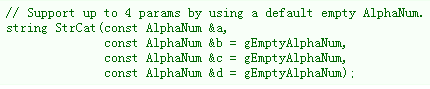
### Default Arguments

**缺省参数：**

我们不允许使用缺省函数参数，如果合适的话，用函数重载模拟它。

一个例外是，如果是.cc文件中（或在未命名空间内）的静态函数，因为函数的使用很局限，这些缺点就体现不出来了。

另一个例外是使用缺省参数来模拟变长参数表时。



### Variable-Length Arrays and alloca()

**变长数组和alloca()**

不允许使用变长数组或alloca()函数。

使用更安全的分配器，如scoped\_ptr/scoped\_array

### Friends

**友元：**

允许有原因的使用友元类和函数。

友元应该应该是定义在同一个文件内的，读者因此不需要去其它文件中查看类的私有成员是如何被使用的。

### Exceptions

**异常**

我们不使用C++的异常。

这个禁止规则也适应C++11中引入的异常相关特性，比如noexcept，std::exception\_ptr和 std::nested\_exception。

### Casting

**转换**

用C++中的static\_cast<>()来进行转换，不要用其他格式转换，比如C风格的强制转换int y = (int)x; int y = int(x);

使用static\_cast进行数值转换，或是显式的将一个类的指针转为它的子类的指针。

使用const\_cast去掉常量特性。

使用reinterpret\_cast进行不安全的指针间转换或整型转指针操作。只有在你清楚操作的含义及可能的后果时才能使用这种转换。

阅读RTTI小节来了解dynamic\_cast的使用。

### Preincrement and Predecrement

**自加和自减**

如果是简单标量（非对象）值，选哪种都行。对于迭代器和其它模板类型，要用前缀形式。

### Use of const

**使用const**

常量的变量、数据成员、方法、和参数会多一层编译期类型检查；这能更快的检查出错误。因此我们强烈的建议你要在任何能用const的地方这么做。

如果函数不会修改传递来的引用或指针参数，这个参数应该是常量。

任何能声明为const的函数都要如此声明。取值函数总该是const。其它函数如果不修改任何成员的值，也不调用任何非常量的方法，且不返回指向数据成员的非常量的指针引用时，也应该声明为const。

如果一个数据成员在构造结束后就不会改变了，就考虑将它声明为常量。

提倡const int \*foo的书写形式，而不是int const \*foo，但不必须，但代码中两种风格的使用要保持一致。

### Integer Types

**整型**

<stdint.h>中定义了如int16\_t、uint32\_t、int64\_t等类型。如果你要求确保整型的大小，你应该用这些类型替代short，unsigned long long等写法。

在C的整型类型中，只有int可以用。合适的话，你最好用标准类型，如size\_t和ptrdiff\_t。

我们经常用int，用在不会特别大的整数上，如循环计数器。这时要用POD类型int。你应该假设int至少有32位，但不要假设它超过32位。如果你需要用64位整数，就用int64\_t或uint64\_t。

对于可能会很大的整数，用int64\_t。

不应该用无符号整型，如uint32\_t，除非你在表示一个位元而不是数字，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为负而使用无符号数。你应该用断言。

**无符号整数**

一些人，包括一些练习书籍的作者，鼓励使用无符号数来表示不会为负的数。这被认为是一种自文档化。但在C中，这种文档化的好处都被它能引起的bug抵消掉了。考虑：



这段代码永远不会停止。有时gcc能找到这个错误并警告你，但通常找不到。同样糟糕的bug会在有符号数和无符号数比较时发生。基本上，C的类型提升机制会导致无符号数表现的和人们期待的不一样。

### Preprocessor Macros

**预处理宏**

谨慎使用宏。用内联函数、枚举量、和const常量来代替宏。

在用宏之前，要仔细考虑是否还有其他的可替代方法。

下面的使用模式能避免使用宏会带来的很多问题；如果你要用宏，就要尽量遵循这些准则：

* 不要在.h文件中定义宏。
* 在马上要用到宏的地方才#define宏，用完立刻#undef。
* 不要#undef一个现有的宏，仅仅因为自己的宏要用它的名字；不如为自己的宏取一个唯一的名字。
* 不要用那些展开后会导致C++构造不稳定的宏，如果非要用，就在文档中说明。
* 不推荐使用##生成函数/类/变量的名字。

### 0 and nullptr/NULL

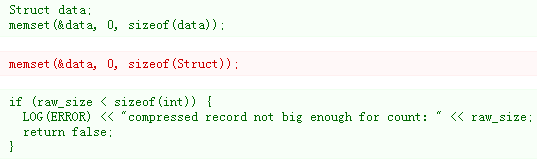
**0和空指针**

0用于整数，0.0用于实数，nullptr（或者NULL）用于指针，’\0’用于字符。

### sizeof

**sizeof:**

相对 sizeof(类型名)，推荐使用sizeof(变量名)。



## Naming

最重要的一致性规则是命名管理。命名的风格能立刻告诉我们这个名字代表的实体到底是什么东西：类型，变量，函数，常量，宏，等等，而不需要我们去寻找该实体的声明。我们头脑中的模式匹配引擎大量依赖于这些命名规则。

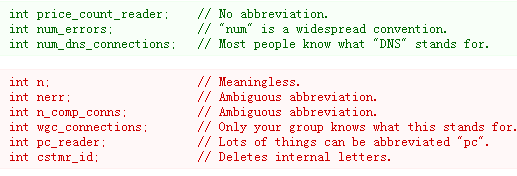
命名规则是很随意的，但我们觉得这方面一致性要比个人喜好重要得多，所以不管你怎么想，规则就是规则。

### General Naming Rules

**通用命名规则**

函数名，变量名，和文件名应该是描述性的；避免使用缩写。

尽可能的给出合理的描述性的命名。不要担心占用空间，能够让新读者立即理解你的代码远比这个重要。不要使用相对项目外的读者来说有歧义或者陌生的命名，不要删除单词中的几个字母来缩写命名。

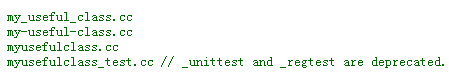


### File Names

**文件命名**

文件名应该全都用小写，中间用“-”或“\_”当分隔符。根据你的项目的惯例。如果没有固定的习惯的话，推荐用“\_”。

可以接受的文件命名示例：



C++文件应该以.cc结尾，头文件以.h结尾。

不要用系统头文件/usr/include中已经存在的文件名，如db.h。

通常来说，让你的文件名非常明确。例如用http\_server\_logs.h就比logs.h更好。通常的情形是文件名成对出现，如foo\_bar.h和foo\_bar.cc，对应类FooBar。

内联函数必须放在.h文件中。如果内联函数很短，就直接放进.h文件中。如果你有很多内联函数，就放到-inl.h文件中。如果一个类有很多内联代码，可以对应三个文件：

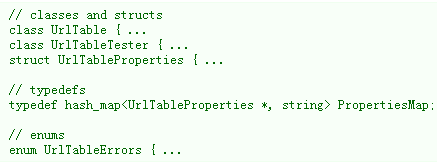


### Type Names

**类型命名**

类型名以大写字母开头，且其中每个词第一个字母都大写，不用下划线：MyExcitingClass，MyExcitingEnum。

所有类型：类，结构体，自定义类型，和枚举，都有着相同的命名惯例。类型名以大写字母开头，且其中每个词也都以大写字母开头。不用下划线。例如：



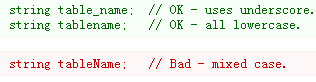
### Variable Names

**变量命名**

变量名都是小写的，每个词之间用下划线连接。类的成员变量名结尾有个下划线。例如：my\_exciting\_local\_varialbe；my\_exciting\_member\_variable\_。

**普通变量命名**

例如：



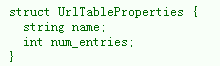
**类数据成员**

数据成员（也被称为实例变量或成员变量）的名字都是小写的，可以像通常的变量名一样带下划线，但结尾总要有个下划线。



**结构体变量**

结构体中的数据成员要像常规变量那么命名，结尾不像类的数据成员那样有下划线。



**全局变量**

对于全局变量没有特别要求，它应该是很少使用的，但如果你要用，考虑加个g\_前缀，或其它能很容易的区分全局变量和局部变量的标记。

### Constant Names

**常量命名**

在名称前加k，其后使用大小写混合：kDaysInAWeek。

所有的编译时常量，不管是局部的，全局的，还是类的一部分，都要遵守一个与其它变量不同的惯例。在名称前加k，后面使用第一个字母大写的各个单词：



### Function Names

**函数命名**

常规函数的名字大小写混合；取值和设值函数要与对应的变量名匹配：MyExcitingFunction()，MyExcitingMethod()，my\_exciting\_member\_variable()，set\_my\_exciting\_member\_variable()。

**常规函数**

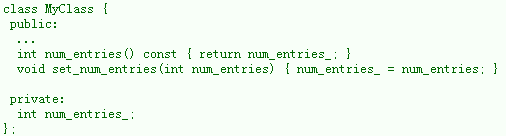
函数名中每个单词首字母都大写，没有下划线。

如果你的函数可能出现错误，你应该在函数名后面加上OrDie。这只适用于发布版本，并且错误可能有原因的发生在正常操作中。



**取值和设置函数**

取值和设值函数应该与相关的变量名匹配。这里摘录一个有着名为num\_entries\_成员变量的类。



其它一些非常短的内联函数也可以全用小写。例如，如果一个函数的开销小到了你在循环中调用它都不需要将结果缓存起来的话，用小写的名字是可以接受的。

### Namespace Names

**命名空间**

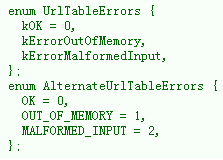
命名空间的名字都是小写的，与项目名，可能还包括目录结构相关：google\_awesome\_project。阅读命名空间小节了解如何命名。

### Enumerator Names

**枚举命名**

枚举的命名应当和常量或宏一致：kEnumName或是ENUM\_NAME。

单独的枚举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 （和 AlternateUrlTableErrors），是类型，因此要大小写混合。



直到2009年1月前，我们的建议是枚举名的风格和宏一致。现在改为采用常量风格的命名。

新代码应该尽量采用常量风格。但也不必将旧代码改成常量风格的，除非旧的名字确实会导致编译错误。

### Macro Names

**宏命名**

你不会真的要定义宏吧？非得用宏的话，要这么命名：MY\_MACRO\_THAT\_SCARES\_SMALL\_CHILDREN。

请看宏小节，通常宏是不应该使用的。

但是，如果真的需要宏，宏名应该全是大写，词之前用下划线分隔。



## Comments

**注释**

### Comment Style

注释风格

用//和/\*\*/都可以，保持一致即可。

你可以用//也可以用/\*\*/；但//更常用。保持注释风格和注释位置与内容的一致性。

### File Comments

**文件注释**

每个文件的开始是版权公告，其后是文件内容描述。

**法律声明和作者信息**

每个文件都应包括许可证信息。为项目选择适合的许可证版本（如Apache2.0、BSD、LGPL、GPL）。

如果你对一个有作者信息的文件进行了大修改，考虑删掉作者信息。

**文件内容**

每个文件都应在开头用注释描述文件内容。

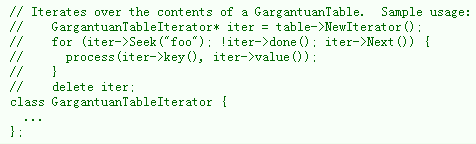
通常来说.h文件会描述文件内声明的类的用途和用法。.cc文件应该包括更多关于实现细节或算法细节讨论等信息。如果你觉得实现细节或算法细节讨论会对阅读.h文件的人有帮助，就可以将这些描述放到.h文件里，但要在.cc中注明文档在.h文件中。

不要在.h和.cc中重复注释。重复的注释容易发生分歧。

### Class Comments

**类注释**

每个类定义都应该有描述用途和用法的注释。



如果你已经在文件开头描述了类的细节，可以简单的标注“见文件开头的完整描述”，但务必确保确实有这类注释。

如果类有任何同步前提，要用文档说明。如果类的实例可能在多线程环境下被访问，要特别注意文档说明多线程使用中的规则和常量使用。

### Function Comments

**函数注释**

函数声明的注释描述了函数的用途；函数定义的注释描述了具体的实现。

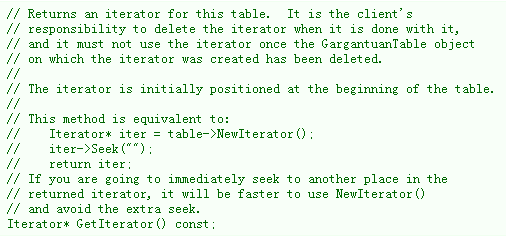
**函数声明**

每个函数声明前面都应有注释来描述函数的用途和用法。这些注释应该是叙述性的（“打开了文件”）而不是命令式的（“打开文件”）；注释是描述函数的，而不是告诉函数去做什么。通常这些注释不会描述函数如何工作的，这些内容留给函数定义处的注释去做。

在函数声明处注释要提及的事情：

* 输入和输出是什么。
* 对于类的成员函数：在函数调用之外对象是否保持对参数的引用，是否会释放这种引用。
* 函数是否分配了必须要调用者释放的内存。
* 函数的参数可不可以是空指针。
* 函数的使用方法上有没有性能隐患。
* 如果函数是可重入的，其同步前提是什么？

这里有个例子：



但不必太啰嗦，或做些显而易见的说明。注意下面的注释没有必要加上“returns false otherwise “，因为这是显然的。



在注释构造函数和析构函数时，记住看你代码的人知道什么是构造函数什么是析构函数，所以如“会销毁此对象“之类的注释都是废话。文档中要说明构造函数对参数做了什么（例如是否取得指针所有权），和析构函数都清理了什么。如果这些都无关紧要，就省略注释。析构函数没有头注释是很常见的。

**函数定义**

如果函数有任何复杂的实现细节，函数定义处都应该有注释来描述。例如你用的编程技巧，实现的大致步骤，或解释为什么你用这种方法实现这个函数而不是用另一种方法。例如你可以提一下为什么函数的前一半需要锁而后一半不需要。

注意你不能只是重复头文件中函数声明处的注释。可以简短的概述一下函数做了什么，但注释的目的是要说明它是怎么做的。

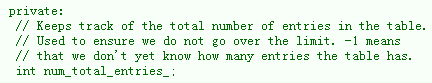
### Variable Comments

**变量注释**

通常变量的名字就应该足够描述清楚变量的用途的了。在某些场合时，也需要额外的注释。

**类数据成员变量**

每个类的数据成员（也被称为实例变量或成员变量）应该有描述其用途的注释。如果这个变量可以取到有特殊含义的值，如空指针或-1，要注释说明。例如：



**全局变量**

所有的全局变量都应该用注释来描述变量的含义和用途。例如：



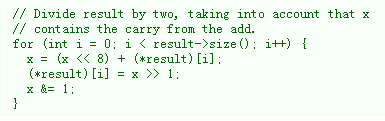
### Implementation Comments

**实现注释**

对于代码中巧妙的、晦涩的、有趣的、或重要的部分，应用注释说明。

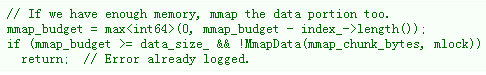
**类数据成员变量**

巧妙的或复杂的代码段应该在前面注释。例如：



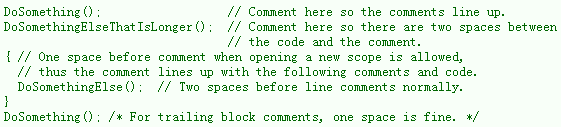
**行注释**

同样的，含义不明显的行应该在行尾注释。这些行尾的注释应该与代码保持2个空格的间隙。如：



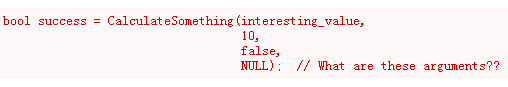
注意有两条注释，一条描述了代码在做什么，另一条提到了在函数因为错误而返回时错误已经被记录在日志中了。

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：

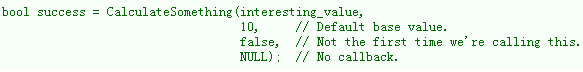


**nullptr/NULL, true/false, 1, 2, 3...**

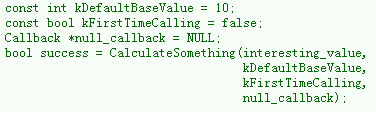
当你向函数传递了一个空指针，布尔变量，或直接量时，应考虑增加一个注释说明它们是什么，或用常量来让你的代码自注释。例如：



与：



或者用常量或自描述型的变量：



**不要做的事**

注意你永远都不应该描述代码本身。要假设看你代码的人比你更懂C++，即便他或她不知道你的用意：



### Punctuation, Spelling and Grammar

**标点，拼写和语法**

要注意标点、拼写和语法；写的好的注释要比写的差的更容易阅读。

注释应该和叙事文本一样可读，有着适当的大小写和标点。在许多情况下，完整的句子要比句子碎片更易阅读。比较短的注释，如在行尾的注释，有时可能不够正式，但你应该和你的风格保持一致。

虽然被别人指出该用分号时却用了逗号多少有些尴尬，但清晰易读的代码还是很重要的。合适的标点、拼写和语法能帮助你实现这一目标。

## Formatting

代码风格与格式确实很随意，但如果所有人都使用同样的风格，项目能更容易进行。个人可能不同意格式规则中的每一方面，一些规则也可能需要一些时间来适应，但所有的项目成员都遵守相同的风格规则是很重要的，这样他们就可以很容易的看懂每个人的代码。

### Line Length

**行宽**

80个字符是最大宽度。

例外：如果注释行包括了一个示例命令或URL的文本超过了80个字符，这行就可以超过80个字符，以便复制剪切。

例外：#include语句如果路径很长的话可能超过80列。试着避免出现这种情况。

例外：你不必关心头文件保护（header guards）是否超过了最大长度。

### Spaces vs. Tabs

**空格还是制表符**

只使用空格，每次缩进2个空格

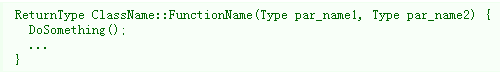
使用用空格进行缩进。不要在你的代码中用tab。你应该将你的编辑器设置为空格替代tab键。

### Function Declarations and Definitions

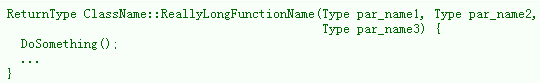
**函数声明和定义**

返回值类型要与函数名要在同一行，参数名如果合适的话也在同一行。

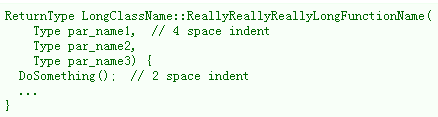
函数像这样：



如果你有太多东西写在了同一行：



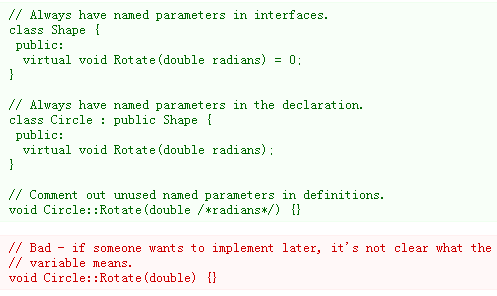
或者如果你连第一个参数都放不进同一行：



一些要记住的要点：

* 如果返回值类型和函数名在一行放不下，在它们之间换行。
* 如果函数定义的返回值类型和函数名之间换行，不需要要缩进
* 左圆括号总要与函数名在同一行。
* 函数名与左圆括号间不要有空格。
* 在圆括号和参数间不要有空格。
* 左大括号总是与最后一个参数同一行。
* 右大括号可以自己在最后一行或（如果其它规则允许）与左大括号在同一行。
* 右圆括号和左大括号之间应该有个空格。
* 所有的参数都要有名字，声明和实现中的名字要相同。
* 如果可能，所有参数都应该对齐。
* 默认缩进是2个空格。
* 换行后的参数缩进4个空格。

如果一些参数没被用到，在函数定义处将参数名注释起来。



### Function Calls

**函数调用**

尽量放在同一行，否则，将实参包含在圆括号中。

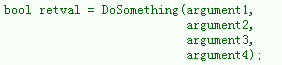
函数调用按下面的格式：



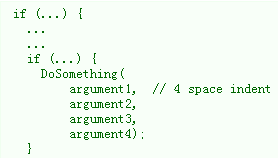
如果实参放不进一行，就将它们分成多行，分出的每行都要和第一个参数对齐。不要在左圆括号后和右圆括号前加空格：



如果函数有很多参数，考虑每行写一个参数，可以增强可读性：



参数可以任意放在后面几行中，每行一个参数：



事实上，如果函数这么长是不满足行宽规则的。

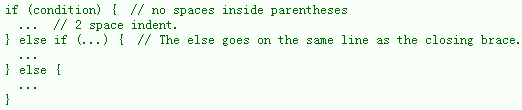
### Conditionals

**条件语句**

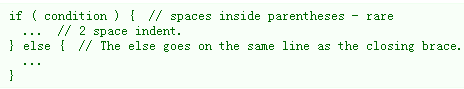
提倡括号内不要有空格。else关键字放在新行中。

有两种基本的条件语句格式都可使用。一种在括号和条件式之间有空格，另一种没有。

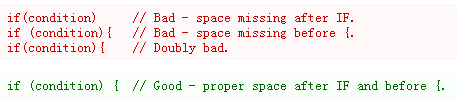
最常见的格式是没有空格的。用哪种都可以，但要保持一致。如果你修改一个文件，就用当前的格式。如果你在写新代码，用项目或目录中其它文件的格式。如果不知道用哪个，你也没有个人倾向的话，不要加空格。



如果你喜欢的话，也可以在括号与条件式之间加空格：



注意任何情况下你都要在if和开括号间放一个空格。你也必须在左大括号和右大括号间放空格。



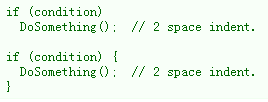
短的条件语句也可以写在同一行内，如果不影响可读性的话。只有在行很短且没有else分句时才能这么用。



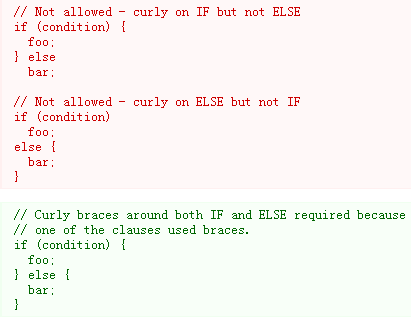
如果if语句包含else的话不许如下这么用：



通常左大括号不需要单独一行，但你喜欢的话也可以；有着复杂的条件或状态的条件语句和循环语句后面跟大括号的话，可能会更有可读性。一些项目要求if后面必须跟着大括号。



但是，如果if-else语句中的一部分用大括号了，另一部分也必须用：



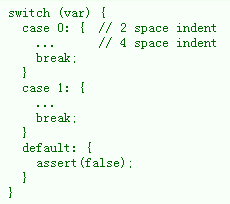
### Loops and Switch Statements

**循环和switch语句**

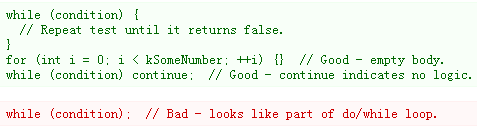
switch语句可以用大括号来分段。空循环体应该用{}或continue。

switch的case块可以用大括号包起来，也可以不用，取决于你的喜好。如果你引入大括号了，应该参照下面的用法。

如果有不满足case条件的枚举值，switch语句应该总是包含一个default case块（如果有输入值没有case条件去处理，编译器会警告）。如果default case永远都不该执行，就放一个assert。



空的循环体应该用{}或continue，而不是单独的一个分号。



### Pointer and Reference Expressions

**指针和引用表达式**

在句点和箭头前后没有空格。指针操作符后面没有尾随的空格。

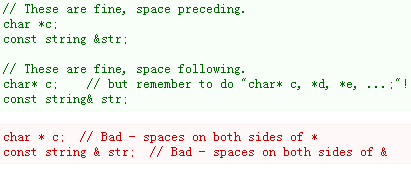
下面是正确的指针和引用表达式：



注意：

* 在访问成员时，句点和箭头的前后没有空格。
* 指针操作符中\*和&后面没有空格。

当声明指针变量或参数时，星号可以挨着类型也可以挨着变量名：



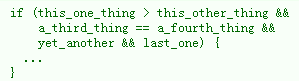
你应该在同一个文件中保持一致，所以在修改一个已有的文件时，用文件原有的风格。

### Boolean Expressions

**布尔表达式**

如果一个布尔表达式比标准行宽还要长，你要像其它情况一样将它分成多行。

这个例子中，与操作符总是在行尾：



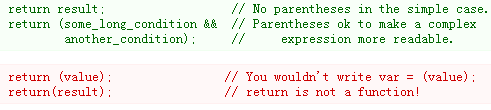
注意本例中代码被分成多行，其中&&操作符都在行尾。

### Return Values

**返回值**

不要用不必要的括号包含返回表达式。

返回表达式return expr;只有在使用x = expr;这种形式时才用括号：

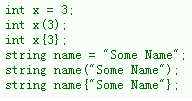


### Variable and Array Initialization

**变量和数组初始化**

你可以用=，()，或者{}。

你可以在=，()和{}中选择；下面的例子都是正确的：

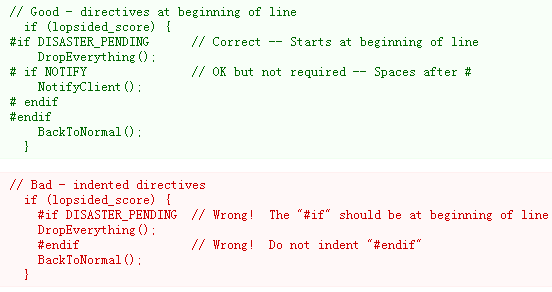


### Preprocessor Directives

**预处理指令**

预处理指令应该总在行首，不要缩进。

即使预处理指令位于缩进的代码块中，它们也不缩进而是总在行首。

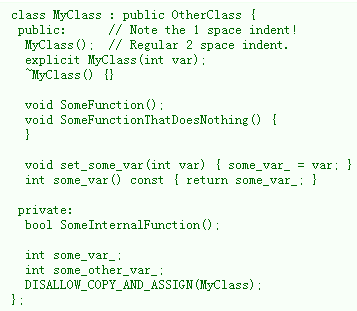


### Class Format

**类格式**

类中各部分按public、protected和private的顺序，每个关键字缩进1个空格。

类声明的基本格式（省略了注释）是：



需要注意的事情：

* 任何基类的名字都应与子类名字在同一行，遵循80列宽的限制。
* public、protected、private关键字应该缩进1个空格。
* 除了第一个出现的关键字，其后的关键字前面应该有个空行。此规则对于较小的类是可选的。
* 不要在这些关键字后留空行。
* public段应在最前，其后是protected段，最后是private段。

### Constructor Initializer Lists

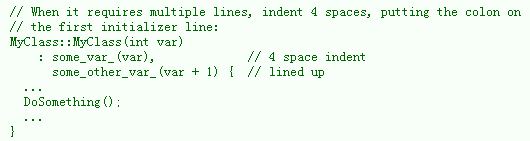
**构造函数初始化列表**

构造函数初始化列表可以都放在一行中也可以分成多行，每行缩进4个空格。

初始化列表有两种可选格式：



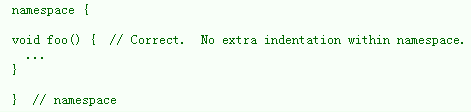
或

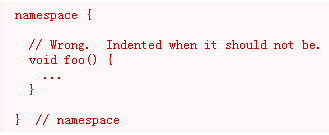


### Namespace Formatting

**命名空间格式**

命名空间的内容不缩进。





声明嵌套命名空间时，每个命名空间单独一行。

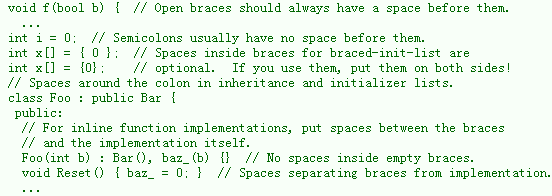


### Horizontal Whitespace

**水平留白**

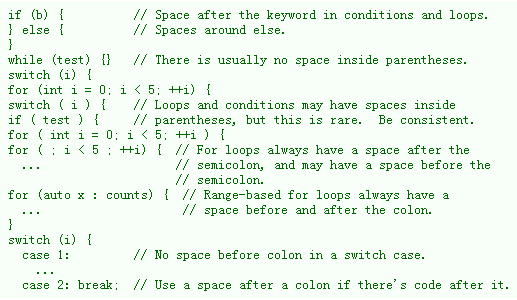
分情况使用水平留白。永远不要在行尾放空格。

**通常情况**

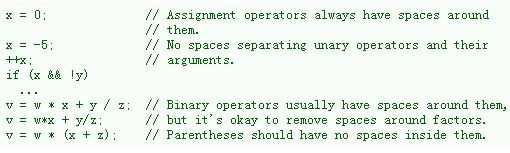


添加冗余的留白会给其他人编辑时造成额外负担，合并行时还需要去除多余的空格。所以：不要引入多余的留白。如果你已经改变完成一行后就去掉行尾的空格，或进行单独的清理操作（推荐在没人使用此文件时进行）。

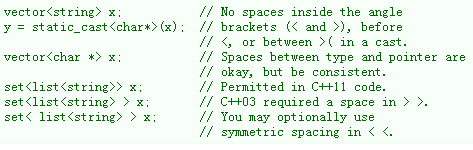
**循环和条件语句**



**操作符**



**模板和转换**



### Vertical Whitespace

**垂直留白**

尽量少用垂直留白。

尤其是不要在函数间留超过一到两个空格，函数首尾不要留空行，函数中也要谨慎使用空行。

使用空行的一些经验法则：

* 函数首尾的空行对提高可读性没有帮助。
* if-else中各块间的空行可能会对此有帮助。