**Google C++ Style Guide**

## Header Files

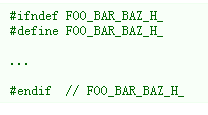
通常每个.cc文件都应该对应有一个.h文件，除非是单元测试或者仅仅包含main函数的比较小的cc文件。

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**

使用前置声明类可以避免不必要的#include。

**定义**：

前置声明用来声明类，函数或者模板，而不需要定义。#include语句经常能够被前置声明所替换，无论符号是否被客户端代码使用。

**优点：**

不必要的#include强制编译器打开更多的文件，处理更多的输入。

不必要的#include一旦头文件发生了改变。代码就要重新编译。

**缺点：**

某些特性如templates，typedefs，缺省参数和使用声明，很难确定其前置声明的形式。

一段代码，特别是涉及到隐式转换时， 决定是使用前置声明还是完整#include是困难的，极端情况下使用前置声明替换#include可能导致代码含义被难以察觉的改变。

前置声明多个符号相比简单的#include头文件显得更冗长啰嗦。

前置声明函数和模板可能妨碍所属头文件对APIs作出兼容性改动，例如扩展参数类型，或者增加带缺省值的模板参数

前置声明命名空间std::中的符号通常导致未定义的行为。

使用前置声明构建代码（比如使用指针成员变量代替对象成员变量）使代码更慢更复杂。

前置声明在实际应用上的效率收益并未得到证明。

**结论：**

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，一个例外是myfile.cc可能会依赖它对应的头文件myfile.h以及其中的前置声明。

### \*Inline Functions

**内联函数**：只有当函数很小，比如10行或者更少时，才定义为内联函数。

**定义：**

你能用一种方式来定义函数，让编译器将它们内联展开，而不是通过通常的函数调用机制来调用它们

**优点：**

当函数很小时，内联展开函数能够产生更有效率的目标代码，将取值和设值函数，以及其它短的，性能关键性的函数内联。

**缺点：**

过度使用内联实际上会使程序变慢。内联函数会导致代码尺寸增加或减少，取决于内联函数的尺寸。内联一个非常小的取值函数通常减少代码尺寸，内联一个非常大的函数会剧烈的增加代码尺寸。现代处理器因为能够更好的利用指令缓存，所以更小的代码通常运行更快。

**结论：**

合理的规则是不要内联一个超过10行的函数。注意析构函数通常比看起来更长，因为隐式的成员变量和基类的析构函数调用。

另外一个有用的规则：内联一个包含循环或者switch语句的函数常常得不偿失，除非通常情况下，循环和switch语句从来不会执行。

重要的是即使是被定义为内联函数也不一定会被内联展开：例如虚函数和递归函数。通常递归函数不应该被内联。内联虚函数将它的定义放到类中的主要原因是为了方便或者为了对它的行为给出文档，比如取值和设值函数。

### The -inl.h Files

**-inl.h文件**：当需要定义复杂的内联函数时，可以使用后缀名为-inl.h的头文件。

一个内联函数需要被定义在一个头文件中，编译器才能在其调用处获得其定义并展开，但是实现代码应该在.cc文件中，我们不喜欢在头文件中有太多代码，除非在可读性和性能上有明显优势。

如果一个内联函数的定义很短逻辑简单，你应该将代码放到.h文件中。例如，取值函数和设置函数就应该包含在类定义中。更复杂的内联函数也可能放在.h中，为了实现和调用的方便，如果这样使.h边的臃肿，你可以将代码放到分离的-inl.h文件中。这样把实现从类定义中分离，当需要时包含对应的-inl.h即可。

另一个使用-inl.h文件的地方是函数模板定义，能提高模板定义的可读性。

不要忘记-inl.h头文件像其他头文件一样要求#define保护。

### \*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。

C/C++函数的参数包含输入输出两种，输入参数也会输出。输入参数通常传值或常量引用，输出参数或者输入/输出参数通常是非常量指针。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。特别是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。

这些不是一个硬性的规则。输入/输出参数（通常是类和结构体）把事情弄复杂了，但是为了和相关函数保持统一可能会要求你屈服。

### \*Names and Order of Includes

**命名和包含顺序**：使用标准的顺序提高可读性，避免隐藏依赖关系：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，详情见下）

2. C库文件

3. C++库文件

4. 其他库文件

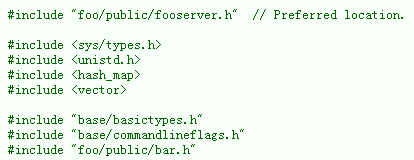
5. 本项目头文件

按照这种排列顺序，如果dir2/foo2.h忽略了任何必要的头文件，编译dir/foo.cc和dir/foo\_test.cc将会中断。因此，这个规则确保编译中断首先被作者发现，而不是其他开发其他包的无辜者。

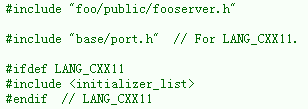
dir/foo.cc和dir2/foo2.h通常在同一个目录中（例如：base/basictypes\_test.cc和base/basictypes.h），但是也可能在不同的目录。

每部分包含的文件应该按字母顺序排列。注意旧的代码可能不符合这个规则，方便的时候应该修复它们。

例如，google-awesome-project/src/foo/internal/fooserver.cc文件中头文件包含可能像这样：



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保护之后。当然，保证你的系统相关代码短小而且小范围的。例如：



## Scoping

### \*Namespaces

**命名空间：**鼓励在.cc文件中使用未命名命名空间。使用命名空间，选择基于项目来命名，可能是它的路径。不需要使用using指示符。不要使用 内联命名空间。

**定义：**

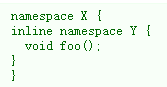
命名空间将全局作用域划分为清晰的有名字的作用域，对防止全局作用域中的命名冲突是非常有用的。

**优点：**

命名空间提供了命名轴线，当然类名也提供了命名轴线。

例如，如果两个不同的项目在全局作用域都有一个名为Foo的类，这个符号可能在编译时或者运行时发生冲突。如果每个项目把它们的代码放到命名空间中，如project1::Foo和project2::Foo非常清晰而不会发生冲突。

内联命名空间自动把它们的名字放到一个封闭的作用域。考虑下面的代码片段，例如：



表达式X::Y::foo()和X::foo()是可以互换的。内联命名空间主要为了跨版本ABI兼容

**缺点：**

命名空间可能令人迷惑的，因为它们在类名提供的命名轴上，增加了另外一个命名轴。

尤其是内联命名空间，令人迷惑的，因为命名实际上不是严格对应定义时的命名空间的。它们仅仅在大的版本策略中有用。

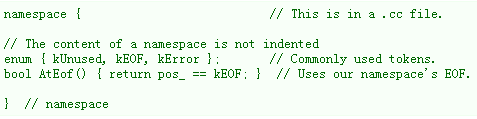
在头文件中使用未命名命名空间容易违法C++中唯一定义原则（One Definition Rule）。

**结论：**

根据下面描述的策略来使用命名空间。在下面的例子中使用注释来标注命名空间结束。

**未命名命名空间**

未命名命名空间在.cc文件中是允许的，而且鼓励这么做，来避免运行时命名冲突：



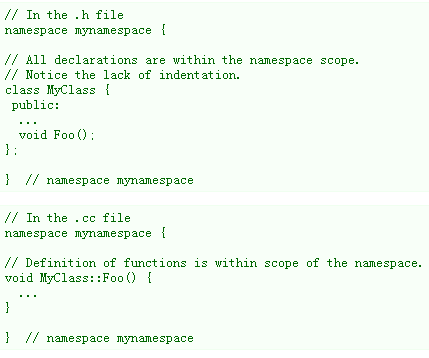
但是，文件作用域内，一些与某个类有关的声明也可以声明为那个类的类型、静态数据成员或是静态函数成员，而不是声明在未命名空间内。

不能在.h文件中使用未命名空间。

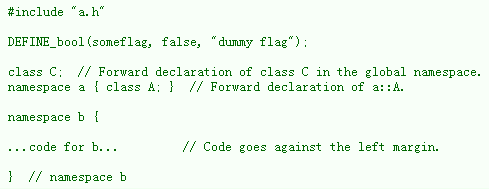
**命名空间**

命名空间应该按以下方式使用：

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为的内容全部包含起来：



通常.cc文件可能包含更多复杂的细节，可能包含需要的其他命名空间的类引用。



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下的实体会导致为定义的行为，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。



注意.h中的别名在引入这个头文件的地方都可见，所以公共的头文件（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，因为要保持公共APIs尽可能的小。

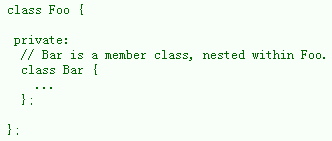
不要使用内联命名空间。

### Nested Classes

嵌套类：当公共的嵌套类作为接口的一部分时，虽然可以直接将他们保持在全局作用域中，但是把它们放在命名空间中是更好的选择。

**定义：**

一个类可以被定义在另一个类里面，也叫成员类。



**优点：**

当嵌套类只在被嵌套类中使用时是有用的。把它作为一个成员放到被嵌套的类中比占用外部作用域中的类名要更好。嵌套类能在被嵌套类中前置声明，然后在.cc文件中定义，避免在被嵌套的类声明中包含嵌套类的定义，因为嵌套类定义通常仅仅只与实现有关。

**缺点：**

嵌套类仅仅能在被嵌套类的定义中能被前置声明，因此，任何头文件包含Foo::Bar\*指针将不得不包含Foo类的完整定义

**结论：**

不要公开嵌套类，除非他们是接口的一部分，比如嵌套类包含某些方法的多个选项。

### \*Nonmember, Static Member, and Global Functions

非成员函数，静态成员函数和全局函数：推荐使用在命名空间内的非成员函数或是静态成员函数来取代全局函数；尽量少用完全的全局函数。

**优点：**

非成员函数和静态成员函数在某些情况下是有用的。把非成员函数放在命名空间中避免污染全局命名空间。

**缺点：**

非成员函数和静态成员函数作为一个新的类的成员时可能更有意义，尤其是当它们访问外部资源或是有着显著的依赖关系时。

**结论：**

有时候定义一个不属于类实例的函数是有用的，或者甚至必须的。这样的一个函数可以是静态成员函数或是非成员函数。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。

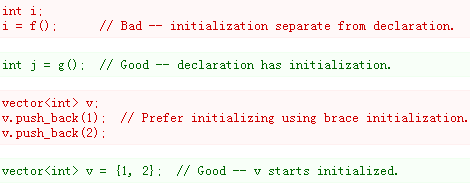
定义在同一个编译单元内的函数在被其它编译单元调用时可能会引入不必要的耦合和链接时依赖；静态成员函数尤其容易受此影响。此时可考虑提取出一个新的类或是将这些函数放到独立库的命名空间内。

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。

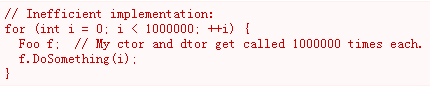
C++允许你在函数的任何定义声明变量。我们提倡尽可能在本地作用域中声明变量，并且尽可能在第一次使用的地方。这样更方法读者找到声明，查看变量类型和初始值。尤其是初始化应该用来取代声明和赋值。例如：



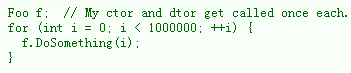
注意GCC能正确实现for (int i = 0; i < 10; ++i)（i的作用范围仅限于循环体内），所以你可以在同一个作用域中重复使用i。在if和while语句中GCC也能正确限制变量的作用域。例如：



有一个警告：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，因为构建和析构顺序不确定，出bug了很难找。

需要持续存在的静态对象，需要持续存在的静态对象，包换全局变量，静态变量，静态成员变量，以及函数中的静态变量，必须是老式的平坦数据类型（POD）：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。

C++中没有指定类的静态变量的构造和初始化顺序，这顺序每次构建都可能改变，这会导致一些bug很难被发现。因此除了禁止全局的类类型变量外，我们也不允许静态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。

同样的，当程序终止，全局变量和静态变量被销毁，不管是否是从main函数中终止或者调用exit函数。析构的调用过程需要和构造的调用顺序相反。因为构造的顺序是不确定的，所以析构的顺序也是不确定的。例如，在程序结束时一个静态变量可能已经被销毁，但代码仍然在运行，也许是其它线程，试图调用这个变量然后失败了。或者一个静态string变量也许已经析构了而另一个变量中还包含对它的引用。

结论是我们只允许POD类型的静态变量。这个准则完全不允许vector（使用C数组）或是string（使用const char[]）。

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）。

**定义：**

可以在构造函数的函数体内进行初始化。

**优点：**

方便打字。不必担心类是否被初始化过了。

**缺点：**

在构造函数中进行工作的问题是：

1.   构造函数没有简单的方法来标记错误，不易于使用异常处理（被禁止的）。

2.   如果操作失败，我们就得到了一个初始化失败而处于不确定状态的对象。

3. 如果在构造函数中调用虚函数，这些调用就会依赖于子类的实现。即使你的类还没有子类，未来对你的类的改动也会悄悄的引入这个问题而带来很多麻烦

如果某人创建了一个这个类的全局变量的实例（即使违背了之前的准则），构造函数中的代码会在主函数之前运行，这也许会违反一些构造代码中隐式的假设。例如此时一些全局符号还没有被初始化

**结论：**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：**如果你的类定义了成员变量，你必须为每个成员变量提供一个类内部初始化器或者写一个默认的构造函数。如果你不定义任何构造函数，编译器会自动生成一个默认构造函数，它可能不会初始化某些变量或者初始化为不合适的值。

**定义：**

默认构造函数会在我们不使用参数来new对象时被调用。在我们用new[]来创建数组时总会调用默认构造函数。类内部成员初始化器意味着使用这样的结构来声明成员变量，比如int count\_ = 17;或者string name\_{"abc"};，而不是int count\_;或者string name\_;。

**优点：**

如果没有提供初始化器，用户定义缺省构造函数来初始化对象。它能保证对象一旦被构造总是被初始化为有效和可使用的状态。它也能保证对象能被初始化为明显不可能的状态，有助于调试。

类内部成员初始化器确保成员变量将被正确的初始化，而不需要在多个构造函数中重复代码。这样当你增加新的成员变量时能减少bug，在一个构造函数中初始化它，却忘了把初始化代码放到其他的构造函数中。

**缺点：**

显式的定义缺省构造函数对于代码编写人员来说是额外的工作

如果成员变量在声明时被初始化，而且在构造函数中成员变量也会被初始化，类内部成员初始化器可能会造成混乱，因此构造函数中的值将会覆盖声明时的值。

**结论：**

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，这应该是更好的方式。

如果你没有其它的构造函数而且没有定义缺省构造函数，编译器将会为你生成一个。编译器产生的构造函数可能不会合理的初始化你的对象。

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：**对只有一个参数的构造函数，使用explicit关键字。

**定义：**通常情况下，如果构造函数有一个参数，它可以用来做隐式转换。例如，如果你定义Foo::Foo(string name)，然后向一个接受参数类型为Foo的函数传递一个字符串，构造函数将会被调用转换字符为并传递对象到你的函数中。这能够方便我们使用，但是也会引发一些你不需要的转换和新对象被创建的问题。使用explicit关键字声明构造函数防止它发生隐式转换。

**优点：**

避免不需要的转换。

**缺点：**

无

**结论：**

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：**仅仅在需要的时候提供拷贝构造函数和赋值操作符重载。否则，使用宏DISALLOW\_COPY\_AND\_ASSIGN来禁止。

**定义：**

拷贝构造函数和赋值操作符被用来创建对象的拷贝。拷贝构造函数在某些情况下被编译器隐式的调用，例如按值传递对象时。

**优点：**

拷贝构造函数使复制对象变得容易。STL容器要所有的内容是可以复制和赋值的。拷贝构造函数比CopyFrom()风格的函数更有效率，因为它们构造的同时还进行了复制。在某些情况下编译器能够省略这些构造操作，这样更容易避免在堆上分配空间。

**缺点：**

隐式拷贝对象是C++中引入bug和性能问题的重要原因。它也降低可读性，因为跟踪按值传递对象比按引用传递要困难，因此很难跟踪到对象在哪里被改变了。

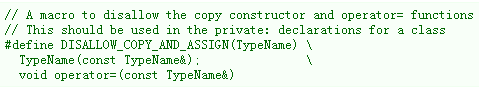
**结论：**

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。在很多情况下，指针或者引用能像一份拷贝一样工作的很好，而且性能更好。例如，你能按引用或者指针代替按值来传递函数参数，你能使用指针代替对象存储在STL容器中。

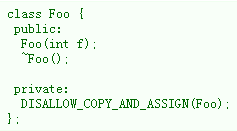
如果你的类需要被复制，优先提供拷贝函数，比如CopyFrom()或者Clone()，而不是拷贝构造函数，因为这些方法不能隐式调用。如果拷贝函数不能满足你的情况（例如性能原因，或者你的类需要按值存储在STL容器中），同时提供拷贝构造函数和赋值操作符。

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：

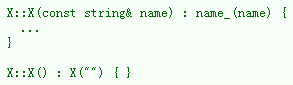


### Delegating and inheriting constructors

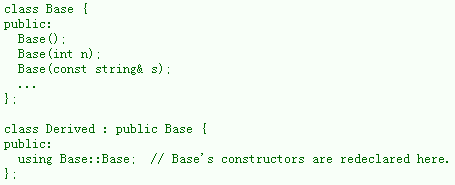
**委托构造函数和派生构造函数：**使用委托构造函数和派生构造函数来减少重复代码。

**定义：**

委托构造函数和派生构造函数是两种不同的特征，在C++11中引入，为了减少构造函数中的代码重复。委托构造函数允许类的一个构造函数去调用另外一个构造函数，使用一种特定的初始化列表语法。例如：



派生构造函数允许派生类通过重新声明派生类构造函数，来直接调用基类的构造函数，就如同调用基类的其他成员函数。如果基类有多个构造器时，这是非常有用的。例如：



相比基类构造函数，派生类构造函数不需要去做任何其他事情的时候，是尤其有用的。

**优点：**

委托构造函数和派生构造函数减少代码重复和引用，能提高可读性。

**缺点：**

通过辅助函数也可能完成委托构造函数的功能。

如果派生类引入了新的成员变量，派生类构造函数可能会引起混乱，因为基类构造函数并不知道他们。

**结论：**

当委托构造函数和派生构造函数，能够减少代码引入和提供代码可读性时，可以使用它们。但你的派生类有新的成员变量时，注意使用派生构造函数。如果你能对派生类的成员变量使用类内部成员变量初始化器，派生构造函数可能仍然适用这种情况。

### \*Structs vs. Classes

**结构体VS类：**仅仅用来承载数据时，使用结构体，其他情况使用类

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。

为了和STL保持一致，使用struct代替class来实现仿函数functors和型别特性traits。

注意结构体和类中的成员变量具有不同的命名规则。

### \*Inheritance

**继承：**通常组合比继承更合适。当使用继承时，使用公有继承，使用public关键字。

**定义：**

当子类从基类继承时，它包含了父类中定义的所有数据和操作定义。事实上派生在C++中被用在两个主要的方面：实现继承，子类继承父类实现代码；接口继承，仅仅方法名称被继承。

**优点：**

实现继承通过重复利用基类代码中已经存在的类型，可以减少代码尺寸。因为继承是在编译器声明，你和编译器能够理解操作并且检测错误。接口继承能够用来强制类暴露特定的API。另外在这种情况下，当类不定义API中必须实现的方法时，编译器能够检测到错误。

**缺点：**

对于实现继承，因为子类中代码实现横跨基类和子类，理解实现会更困难。子类不能覆盖基类中非虚拟函数，因此子类不能改变实现。基类也定义了一些数据成员，还要指定基类的物理布局。

**结论：**

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可以子类化Foo。

如果需要的话，确保析构函数是虚拟的。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。注意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：**仅仅少数场合多重继承是有用的。当有实现的基类只有一个时，我们允许多重继承；所有其他基类必须是纯接口类，并且名字使用Interface后缀。

**定义：**

多重继承允许子类有超过一个基类。我们将基类分为纯接口类和实现类。

**优点：**

比起单继承，多重继承可能让你重复利用更多的代码。

**缺点：**

仅仅非常少的场合多重继承是有用的。某种场合下如果用多重继承的解决方案，你通常也能找到一种不同的、更明确和更清晰的解决方案。

**结论：**

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。

注意：Windows不适用这个规则。

### \*Interfaces

**接口：**满足某些特定的条件，但是又不是必须的类，可以以Interface结尾

**定义：**

一个类是否是纯接口需要满足下面的需求：

* 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）。

**优点：**

用Interface后缀来标识一个类能让其它人知道他们禁止向其中添加方法实现或非静态的数据成员。这在多重继承中尤其重要。另外，接口的概念已经被Java程序员所熟知了。

**缺点：**

Interface后缀增加了类名字的长度，这会令人难以阅读和理解。此外，接口特性作为实现细节不应该被暴露给客户端。

**结论：**

一个类只有在满足上述要求时才能用Interface后缀。但是我们不需要相反的规则：满足上述条件的类不要求以Interface结尾。

### \*Operator Overloading

**操作符重载：**除非在特殊场合，不要使用操作符重载。

**定义：**

一个类可以定义诸如“+“、”/“等操作符，就像内建数据类型一样。

**优点：**

能让代码看起来更直观，因为类与内建类型有着相同的表现（比如int）。重载运算符使 Equals()、Add() 等函数名黯然失色。为了让一些函数模板正确工作，你可能必须定义某些操作符。

**缺点：**

操作符重载能令代码更直观的同时，它也有着一些缺点：

* 这会误导我们的直觉，使我们认为代价昂贵的操作其实是廉价的内建操作。
* 想发现重载操作的调用点变得更困难了。查找 Equals() 发生的位置要比查找“ == “操作符来的容易得多。
* 一些操作符也工作在指针上，这更容易引入bug。Foo + 4是一种操作，而&Foo + 4却是完全不同的另一种操作。编译器是不会抱怨这种事的，这导致很难调试。

**结论：**

一般来说不要重载操作符。尤其是赋值操作符（operator=）隐蔽性很高，应该避免重载它。你可以定义像Equals()和CopyFrom() 这样的函数。同样的，一个类只要有可能被前向声明，无论如何不要重载危险的一元操作符&。

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

在类中使用特定的声明顺序：公有的要在私有的前面，方法要在数据成员前面，等等。

你的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。

不要在类定义中将很大的方法定义为内联的。通常，只有琐碎的或性能关键性的，以及非常短的方法才能被定义为内联的。

### \*Write Short Functions

**尽量编写短而专一的函数**

我们认识到有时长函数也是需要的，所以在函数的长度上没有硬性限制。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。

即使你的长函数工作得非常好，几个月的修改后可能会引入新的行为，最后导致难以察觉的bug。保持你的函数短而简洁，能让其它人更容易阅读和修改。

你工作时可能会发现长而且复杂的函数。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。

## Google-Specific Magic

我们使用许多窍门和实用技巧来让C++代码更强健，我们使用C ++的许多方法也可能不同于你在其它地方见到的。

### Ownership and Smart Pointers

### cpplint

## Other C++ Features

### \*Reference Arguments

**引用参数:** 所有按引用传递的参数必须被标记为const。

**定义：**

在C中，如果一个函数需要修改一个变量，就必须使用指针参数，如int foo(int \*pval)。在C++中，函数还可以声明一个引用参数：int foo(int &val)。

**优点：**

将参数定义为引用能避免像(\*pval)++这样丑陋的代码。在一些应用场合必须使用引用，比如拷贝构造函数。相比指针更清晰，不会接受空指针参数。

**缺点：**

引用在语法上像值，语义上像指针，这可能会令人迷惑。

**结论：**

函数参数表中所有的引用都必须使用const修饰：



事实上这在谷歌代码中是一种硬性约定：输入参数是值或常量引用，输出参数是指针。输入参数也可以是常量指针，但不允许使用非常量的引用参数，除非有预定。

但是也有一些场合用const T\*当输入参数要比 const T&好。例如：

* 你想传递进空指针。
* 这个函数要保存指向输入的指针或引用

要记住大多数情况下输入参数都应该是常量引用const T&。用常量指针const T\*的时候就是告诉读者输入要被特殊对待了。所以如果你要用常量指针const T\*就要有个明确的理由；否则就会迷惑读者去寻找不存在的例外情况。

### Rvalue references

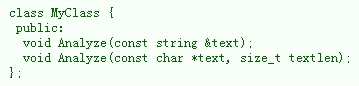
右值引用

### \*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。

**定义：**

你可以写一个参数为const string&的函数，再写一个参数为const char\*的同名函数重载它。



**优点：**

通过允许函数名字相同但参数不同，重载能令代码更直观。对于模板化的代码重载也是必要的，对调用者来说也更方便。

**缺点：**

如果一个函数只被重载之后通过参数来区分，读者可能不得不去理解C++的复杂的匹配原则才能知道发生了什么。而如果一个派生类只重写了某些函数，许多人也会被继承的语义所迷惑。

**结论：**

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，除非是下面解释的几种情况。如果合适的话，用函数重载模拟它。

**优点：**

通常你写的函数可能会用到很多的缺省值，但偶尔你也会修改这些缺省值，不需要为了这些罕见的情况定义很多的函数，用缺省参数就能很轻松的做到这点。相比重载这些函数，缺省参数语法更清晰，引用更少，而且可以清晰的区分必选和可选参数。

**缺点：**

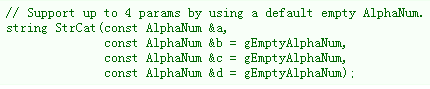
存在缺省参数时很难写函数指针，因为函数的形式和调用的形式经常不匹配。给一个已经存在的函数增加一个默认参数就会改变它的类型，这会导致使用函数地址的代码出现问题。增加一个重载的函数能避免这些问题。此外，缺省参数可能会导致笨重的代码，因为每次调用时参数都会被复制，与重载函数截然相反，“缺省的”仅出现在函数的定义中。

**决定：**

上述的缺点不那么夸张，仍然超过了缺省参数相比函数重载带来的好处。因此除非下面描述的情况，我们要求所有参数必须明确指定。

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。

**优点：**

变长数组语法很自然，且和alloca()一样有着很高的效率。

**缺点：**

变长数组和alloca()函数不是标准C++的一部分。更重要的是，它们在栈上分配的空间与实际的数据有关，可能会导致一些很难发现的内存改写问题：“在我的机器上运行正常，但发布出去就莫名其妙的退出。

**结论：**

使用更安全的分配器，如scoped\_ptr/scoped\_array

### \*Friends

**友元：**我们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。一种很常见的使用友元的方法是令FooBuilder类为Foo类的友元，这样前者就可以正确的构建Foo的内部状态，而不用将这个内部状态暴露给外界。有时将单元测试类设置为待测试类的友元会很有用。

友元扩展而不是打破了类的封装的界限。在一些情况下只对一个其它类公开访问权限，要比公开一个成员更好。但是大多数类还是应该只通过公开成员来与其它类交互。

### \*Exceptions

**异常：**我们不使用C++的异常。

**优点：**

* 异常允许上层应用程序决定如何处理底层嵌套函数出现的“不该发生“的错误，而不是用模糊且易错的错误码记录。
* 大多数其它现代语言中都包含异常。在C++中使用异常有助于与Python、Java和其它与C++类似的语言的统一。
* 一些第三方C++库使用异常，如果禁用异常的话就很难集成这些库。
* 异常是处理构造函数失败的唯一方法。我们能用工厂函数或Init()来模拟这个特性，但这些分别需要进行堆的分配或是新增一个“无效”状态。
* 在测试框架内使用异常真是得心应手。

**缺点：**

* 在已经存在的函数中增加throw语句时，你必须检查所有调用点。所有调用点必须至少做到基本的异常安全保证，否则永远捕捉不到这个异常，而只能“笑着“面对程序终止的结果。例如，如果f()调用了g()，g()调用了h()，h抛出的异常被f捕捉，那g就需要当心是否未妥善处理。
* 更普遍地，使用异常导致只看代码很难评估程序的控制流：函数可能会在你不期望的位置返回。这加大了维护和调试的难度。你可以通过规定异常使用的方式和位置来最小化开销，但是让开发人员必须掌握并理解这些规定带来的代价更大。
* 异常安全需要同时使用RAII和不同编程实践。想轻松写出正确的异常安全的代码需要很多支持机制。另外，要避免要求读者理解整个调用体系，异常安全的代码必须将写入持久化状态的逻辑隔离到“提交“阶段。这么做既有好处也有代价（也许你被迫故意写出模糊的代码来隔离提交）。允许异常会强迫我们总是承担这些开销，就算我们觉得不划算。
* 开启异常会增加每个二进制文件的数据，增加编译时间（可能是轻微的），还有可能增加地址空间压力。
* 允许使用异常，可能会怂恿开发者在不恰当的场合抛出异常或是在不安全的地方从异常中恢复。例如，用户输入不合法不应该抛出异常。如果我们要完全列出这些约束, 这份风格指南会长出很多。

**结论：**

表面上看，使用异常利大于弊，尤其是在新项目中。但对于现有的代码，引入异常会牵连到所有相关的代码。如果新项目允许异常，当与现有的无异常的代码整合的时候也会出问题。因为谷歌现有的大多数C++代码都不包含异常，引入能产生异常的新代码时比较困难。

鉴于谷歌现有的代码不接受异常，使用异常的开销比在新项目中使用要大一些。转换过程缓慢且易错。我们相信异常机制的替代品，如错误码和断言，不会带来显著的负担。

我们不是基于哲学或道德层面反对使用异常，而是在实践的基础上。因为在谷歌我们希望用我们的开源项目，而这些项目很难使用异常，所以我们也不建议在谷歌开源项目中使用异常。如果我们不得不从头开始，事情可能就不同了。

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。

对于Windows代码来说，不适用这个规则。

### Run-Time Type Information (RTTI)

**运行时类型信息：**避免使用RTTI

### \*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);

**定义：**

C++中引入了与C不同的转换系统，它用多种不同的转换操作符来区分各种操作。

**优点：**

C风格的转换的主要问题是操作的含义太模糊了：有时是数值转换（如(int)3.5），有时是类型转换（如(int)”hello”）；C++的转换就避免了这一点 。另外，C++的转换更容易被搜索到。

**缺点：**

语法令人不快。

**结论：**

不要使用C风格的转换。改用C++风格的转换。

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。

### Streams

### \*Preincrement and Predecrement

**自加和自减：**对于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。

**定义：**

一个变量在自加（++i或i++）或自减（--i或i--），而没有用到表达式的值时，必须确定到底用哪种形式。

**优点：**

忽略返回值时，前缀形式（++i）至少不会比后缀形式（i++）效率低，常常更高。这是因为后缀自加（或自减）需要保存i的原始值作为表达式的值。如果i是迭代器或其它非标量类型，复制操作代价可能会很昂贵。既然忽略表达式的值时两种表示方法都一样，为什么不一直用前缀形式呢？

**缺点：**

C的传统写法是用后缀形式，尤其是在循环中。有人说后缀形式容易阅读，因为名词i放到了动词++的前面，和英语写法类似。

**结论：**

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：**能用const的时候都要用const。在C++11中，某些情况下constexpr是更好的选择。

**定义：**

声明变量和参数时，可以在前面加上关键字const以表示这个变量的值不会被改变（例如const int foo）。类的成员函数也可以加const，表示这个函数不会改变类的成员变量的状态。例如：class Foo { int Bar(char c) const; };

**优点：**

容易让人理解变量的使用方式。允许编译器更好的进行类型检查，以及能产生更好的代码。帮助人们相信程序的正确性，因为他们知道调用的函数不会修改他们的变量。帮助人们知道在多线程环境下什么函数不需要互斥锁也是安全的。

**缺点：**

const就像病毒：如果你向函数传递了常量参数，函数的原型中就需要写上const（否则需要用const\_cast转换变量）。在调用库函数时这确实是个问题。

**结论：**

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。

把const放在哪：

有人更喜欢int const \*foo的形式，而不是const int \*foo。他们说这么更有可读性，因为这种做法更有一致性：它满足这样一条规则，const应该总是跟着它修饰的对象。但是，这种一致性的说法并不适用于深度嵌套的指针表达式很少的代码库，因为大多数常量表达式只有一个const符，一致性的价值体现的不明显。这种情况下不用维护一致性。有观点认为把const放前面更容易阅读，因为它满足英语语法：把形容性（const）放到名字(int）前。

总之，我们鼓励将const放前面，但不必须。但代码中两种风格的使用要保持一致。

### Use of constexpr

### \*Integer Types

**整型：**C++的内建整数类型中，只使用int。如果程序中需要用到其它大小的变量，用<stdint.h>中带精度的整数类型，如int16\_t。如果你的变量可能大于或者等于2^31，使用64位类型int64\_t。注意你的值不会超过int的取值范围，用于中间计算可能要求大的数据类型，如何有疑虑，选择更大的类型。

**定义：**

C++没有指定各个整数类型的大小。通常假定短整型是16位的，整型是32位的，长整型是32位的，长长整型是64位的

**优点：**

声明一致。

**缺点：**

整型的大小依赖于编译器和架构。

**结论：**

<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。

你不应该用无符号整型，如uint32\_t，除非你在表示一个位组而不是数字，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。

所以，用断言来提醒变量是非负的。不要用无符号类型。

### 64-bit Portability

### \*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。

使用宏就意味着你看到的代码和编译器最终看到的代码是不一样的。这会带来难以预料的行为，特别是因为宏是全局作用域。

幸运的是，在C++中宏不像在C中那么重要 。使用内联函数来代替宏实现性能关键性的代码，不要用宏来存储一个常数，改用常量。不要用宏来为一个名字很长的变量起个缩略名，改用引用。不要用宏来进行条件编译，改用...，好吧，不要使用预处理宏（除非是用#define来避免多次引入头文件）。用宏会导致很难进行测试。

宏能做很多不可替代的工作，你可以在代码库，尤其是底层代码库中看到这些工作。而且宏的一些特性（如字符串化，文本串连，等等）无法通过语言特征来实现。但在用宏之前，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。

0用于整数0.0用于浮点数，这是毫无疑问的。

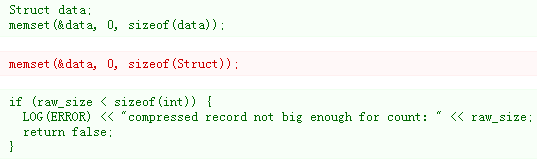
对于指针（地址值），可以选择用0，NULL（C++11中还能用nullptr）。对于允许C++11特性的项目，要用nullptr。对于C++03的项目，推荐用NULL，因为它看起来像指针。事实上，一些C++编译器提供了特殊的NULL定义，可以给出有帮助的警告信息，尤其是sizeof(NULL)不等于sizeof(0)的情况。

‘\0’用于字符。不仅类型正确，可读性也强。

### \*sizeof

**sizeof:**相对 sizeof(类型名)，推荐使用sizeof(变量名)。

当计算一个特定变量的大小时使用sizeof(变量名)是因为如果变量的类型改变了，它也能跟着更新。sizeof(类型名)可以使用在与特定变量无关的代码中，比如一个C++类型的变量不方便使用。



### auto

**自动识别类型**

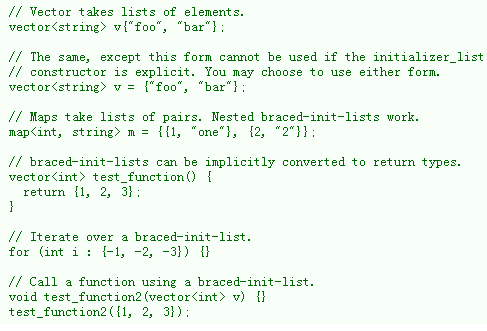
### Brace Initialization

**大括号初始化：**你可以使用大括号初始化

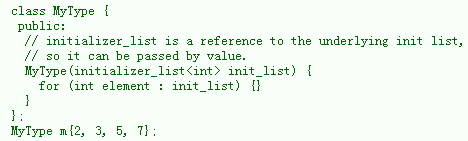
在C++03中，聚合类型（数组和结构体没有构造函数）能够使用大括号初始化。



在C++11中，语法被扩展到其他数据类型。大括号初始化形式被叫做*braced-init-list*。下面有几个例子。



用户数据类型也能使用初始化列表initializer\_list来定义构造函数，它自动创建*braced-init-list*。



### Lambda expressions

lambda表达式

### Boost

Boost

### C++11

## 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。

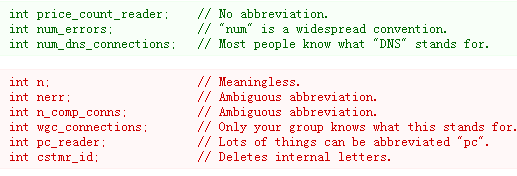
命名规则是很随意的，但我们觉得这方面一致性要比个人喜好重要得多，所以不管你怎么想，规则就是规则。

### \*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原因比这个重要。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。

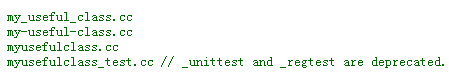


### \*File Names

**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：

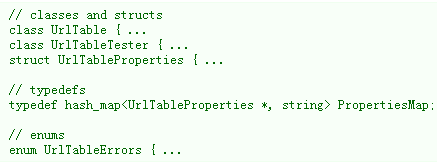


### \*Type Names

**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：



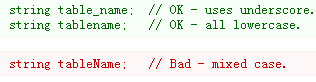
### \*Variable Names

**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：



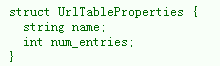
**类数据成员**

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**

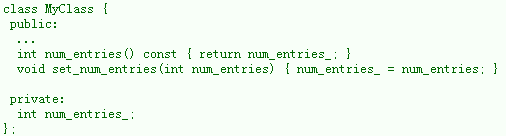
函数名中每个单词首字母都大写，没有下划线。

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**

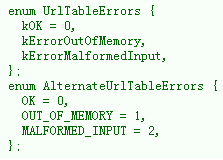
命名空间的名字都是小写的，与项目名，可能还包括目录结构相关：google\_awesome\_project。阅读命名空间小节了解如何命名。

### \*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。这导致了一些枚举与宏的命名冲突。因此，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。



### \*Exceptions to Naming Rules

命名规则中的例外

如果你命名的实体与已有C/C++实体类似，可参考现有命名规则。

函数名，参考open()

uint参考自定义类型

结构体或类，参考pos

STL风格实体；参考STL命名规则

常量，参考INT\_MAX

## Comments

注释尽管写起来很痛苦，但在保持代码的可读性方面至关重要。下面的规则描述了你应该在哪注释什么。但要记住：注释很重要，但最好代码本身就是自注释的。给类型和变量起有意义的名字要比用难懂的名字导致必须得用注释解释强得多。

在写注释时，目标是你的观众：下一个需要看懂你代码的作者。要好好写，下一个目标观众可能就是你！

### \*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。

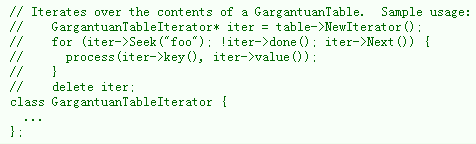
通常来说.h文件会描述文件内声明的类的用途和用法。.cc文件应该包括更多关于实现细节或算法细节讨论等信息。如果你觉得实现细节或算法细节讨论会对阅读.h文件的人有帮助，就可以将这些描述放到.h文件里，但要在.cc中注明文档在.h文件中。

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。

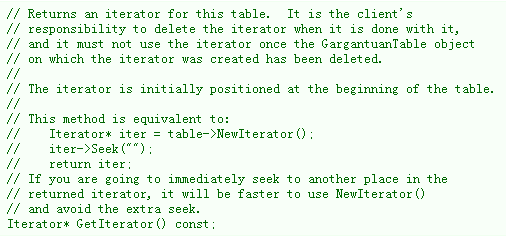
**函数声明**

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。

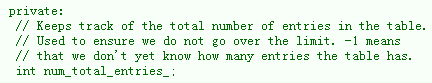
### \*Variable Comments

**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：



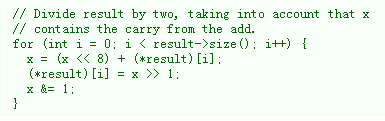
### \*Implementation Comments

**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。

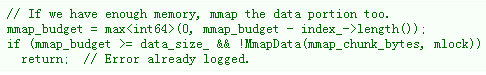
**类数据成员变量**

巧妙的或复杂的代码段应该在前面注释。例如：



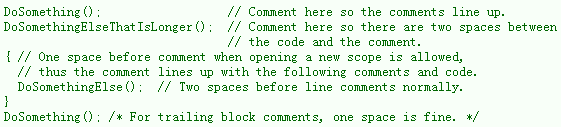
**行注释**

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：



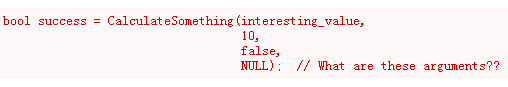
注意有两条注释，一条描述了代码在做什么，另一条提到了在函数因为错误而返回时错误已经被记录在日志中了。

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：

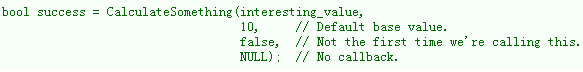


**nullptr/NULL, true/false, 1, 2, 3...**

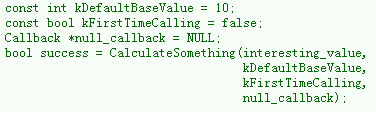
当你向函数传递了一个空指针，布尔变量，或直接量时，应考虑增加一个注释说明它们是什么，或用常量来让你的代码自注释。例如：



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。

### TODO Comments

**TODO注释**

将TODO注释用在临时性的、短期解决方案、或足够好但还不完美的代码上。

TODO注释应该包括全大写的“TODO”，接着是能提供最多信息的人的名字，邮箱地址，或其它个人信息。冒号是可选的。主要目的是能让添加注释的人（或可提供最多信息的人）按统一的TODO格式搜索到。TODO注释不表示相关的人承诺会修复此问题。因此当你新建了一个TODO注释，它给出的总是你的名字。



如果你的TODO注释是“未来某时会做某事”的格式，要确保你要么包括了一个具体的时间（“2005年11月修复”）要么包括具体的事件（“当所有客户端都能处理XML响应后移除此代码”）。

### Deprecation Comments

**过期的注释**

用DEPRECATED注释来标记过期的接口。

你可以通过给某接口加上DEPRECATED的注释来标记其为过期的。可以在接口声明之前注释，也可在同一行注释。

在DEPRECATED单词后的括号里写上你的名字、邮箱或其它个人信息。

过期内容的注释必须包括简单清晰的指示用于指导用户修正调用。在C++中，你可以将一个过期的函数实现为内联函数，在其中调用新的接口。

用DEPRECATED标记一个接口不会魔法般的导致任何调用被改变。如果你想要用户真的停止使用过期的调用，你需要自己修复调用或招几个人来帮你。

新代码中不应包含对过期接口的调用。改为使用新接口。如果你看不懂提示，就找添加了这个过期说明的人，然后请求他帮你使用新接口。

## 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**

你每行代码的文本最多只允许80个字符宽。

我们承认这条规则有争议，但有这么多现存的代码已经遵守它了，我们感觉一致性更重要。

**优点：**

喜欢这条规则的人认为强迫他们改变编辑器窗口大小很野蛮，而且更长的代码也没什么必要。一些人习惯多个代码窗口挨着排，因此没办法让他们的窗口再宽一点点了。人们都将他们的工作环境设为某个特定的窗口宽度，而80列就是传统的标准。为什么要改变它？

**缺点：**

支持改变此规则的人认为更宽的行能令代码更有可读性。80列的限制可以回溯到古老的1960年代的主机；现代机器有着更宽的屏幕，能轻易的显示更宽的行。

**结论：**

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。

### \*Non-ASCII Characters

**非ASCII字符**

要少用非ASCII字符，即使用的话也必须用UTF-8格式。

你不应将用户界面文本硬编码进源文件中，即使是英文，这样就不怎么需要用非ASCII字符了。但是，有些场合就适合将非ASCII单词放进代码中。例如，如果你的代码要解析国外来源的数据文件，将在这些文件中充当定界符的非ASCII字符串硬编码可能会比较合适。更常见的情况是单元测试代码（不需要本地化）可能会包含非ASCII字符。这些场合你都应该用UTF-8格式，因为这种编码被大多数能处理不止ASCII这一种编码的工具所支持。16进制编码也可以，在能增强可读性的地方推荐这种编码，例如，“\xEF\xBB\xBF”是Unicode零宽度无间断空格字符，而如果直接按UTF-8格式加进源文件中就看不到这个字符了。

使用包含转移字符\uXXXX的u8 前缀来保证字符串编码为UTF-8。不要用它来把包含非ASCII字符的字符串编码为UTF-8，因为不能解释UTF-8编码源文件的编译器会产生错误。

你不应该使用C++11中的字符类型char16\_t和char32\_t。因为它们是非UTF-8文本。同样的原因，你也不应该使用wchar\_t（除非你正在写的代码需要和使用wchar\_t的 Windows API交互）。

### \*Spaces vs. Tabs

**空格还是制表符**

只使用空格，每次缩进2个空格

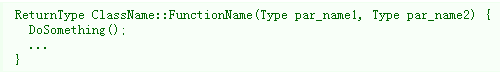
我们用空格进行缩进。不要在你的代码中用tab。你应该将你的编辑器设置为空格替代tab键。

### \*Function Declarations and Definitions

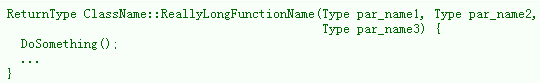
**函数声明和定义**

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。

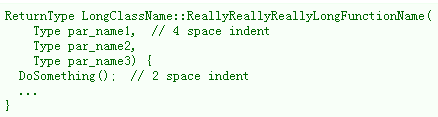
函数像这样：



如果你有太多东西写在了同一行：



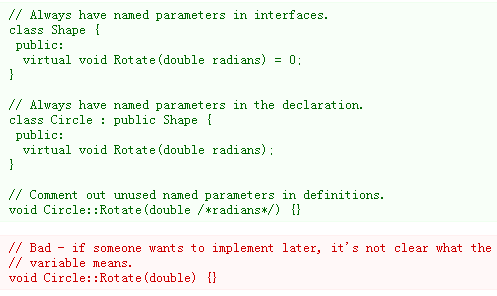
或者如果你连第一个参数都放不进同一行：



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。

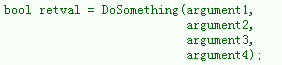
函数调用按下面的格式：



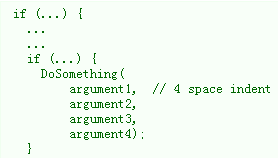
如果实参放不进一行，就将它们分成多行，分出的每行都要和第一个参数对齐。不要在左圆括号后和右圆括号前加空格：



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。

### Braced Initializer Lists

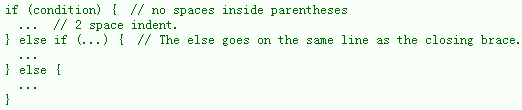
### \*Conditionals

**条件语句**

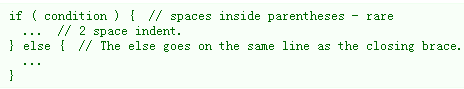
提倡括号内不要有空格。else关键字放在新行中。

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。

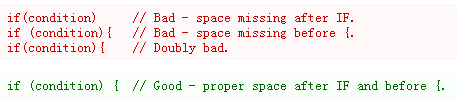
最常见的格式是没有空格的。用哪种都可以，但要保持一致。如果你修改一个文件，就用当前的格式。如果你在写新代码，用项目或目录中其它文件的格式。如果不知道用哪个，你也没有个人倾向的话，不要加空格。



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。



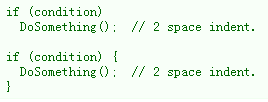
短的条件语句也可以写在同一行内，如果不影响可读性的话。只有在行很短且没有else分句时才能这么用。



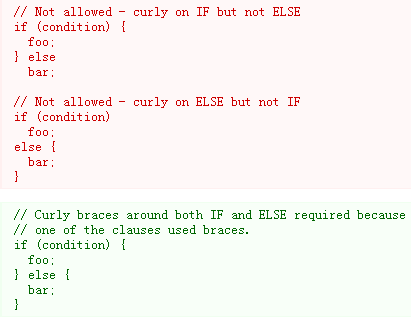
如果if语句包含else的话不许如下这么用：



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：



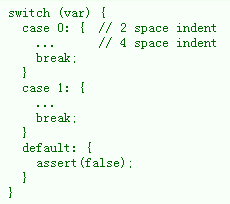
### \*Loops and Switch Statements

**循环和switch语句**

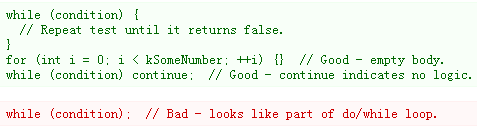
switch语句可以用大括号来分段。空循环体应该用{}或continue。

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。

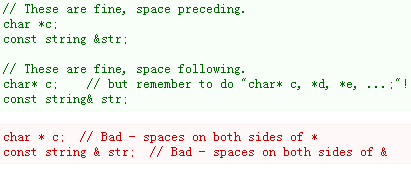
下面是正确的指针和引用表达式：



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：



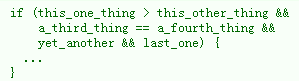
你应该在同一个文件中保持一致，所以在修改一个已有的文件时，用文件原有的风格。

### \*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：



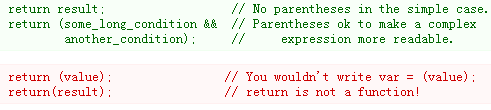
注意本例中代码被分成多行，其中&&操作符都在行尾。尽管也可以将所有操作符都对齐到行首，本例中的用法在谷歌的代码中更常见。这样可以方便插入括号，适当使用对提升可读性很有帮助。同样注意你应该总是用符号表示的操作符，如&&和~，而不是用单词表示如and和compl。

### \*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：

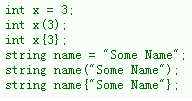


### \*Variable and Array Initialization

**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：

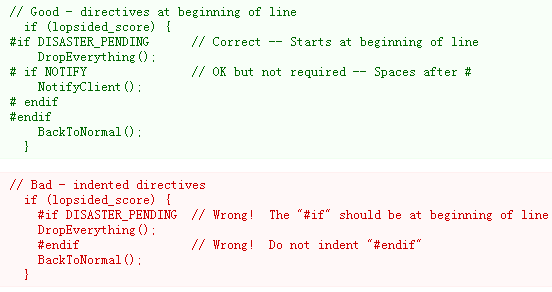


### \*Preprocessor Directives

**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。

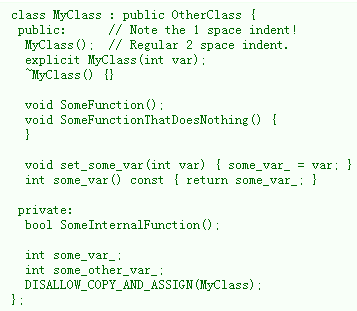


### \*Class Format

**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

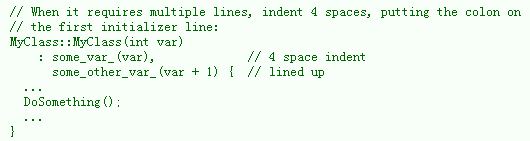
**构造函数初始化列表**

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：



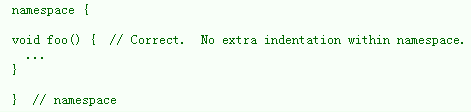
或

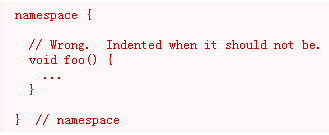


### \*Namespace Formatting

**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。

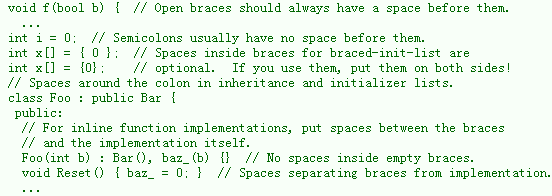


### \*Horizontal Whitespace

**水平留白**

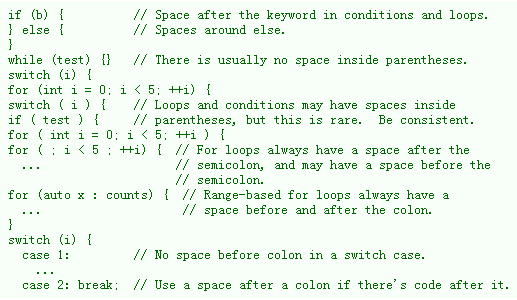
分情况使用水平留白。永远不要在行尾放空格。

**通常情况**

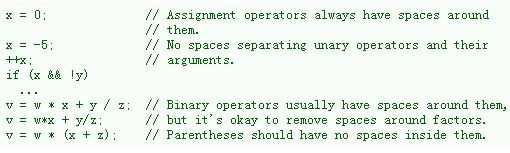


添加冗余的留白会给其他人编辑时造成额外负担，合并行时还需要去除多余的空格。所以：不要引入多余的留白。如果你已经改变完成一行后就去掉行尾的空格，或进行单独的清理操作（推荐在没人使用此文件时进行）。

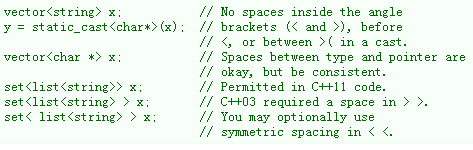
**循环和条件语句**



**操作符**



**模板和转换**



### \*Vertical Whitespace

**垂直留白**

尽量少用垂直留白。

这不仅仅是规则而是原则问题：不必要的时候不要用空行。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。

基本原则是：同一屏幕内能显示的代码越多，理解掌握程序的控制流越容易。当然，思路过分发散，导致代码过于密集，也会导致可读性下降，所以需要你自己把握。但通常来说要尽量少用垂直留白。

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。

## Exceptions to the Rules

**规则之外的例外**

上面叙述的代码习惯是强制性的。但是像所有的良好规则一样，这些规则有时也有例外，此处就讨论这些例外。

### Existing Non-conformant Code

**现有不合规范的代码**

在处理不符合本风格规范的代码时，你可以不遵守上面的规则。

如果你在修改不符合本规范的代码时，你可能需要放弃这些规则，来和代码中原有的习惯保持一致。如果你困惑该怎么做，就去问原作者或当前负责此代码的人。记得要保持局部一致性。

### Windows Code

**Windows代码**

Windows程序员已经发展出了他们自己的编程习惯，主要源于Windows头文件和其它微软代码中的习惯。我们想让任何人都能顺利理解你的代码，所以我们为在任何平台写C++代码的人都准备了一个单独的规范集合。

## Parting Words

**结束语**

运用常识，并保持一致。

如果你在编辑代码，花一些时间看看周围的代码再决定用哪种风格。如果它们在if分句前后留有空白，你也应该这么做。如果它们的注释用星号组成的矩形框包起来，你也这么做。

风格指南的重点是提供一个通用的编程规范，这样人们就可以专注于实现内容，而不是表达方式。我们展示了全局的风格，但局部风格也很重要。如果你加进一个文件的代码与原有代码完全不同，这种不连续会破坏读者阅读的节奏。试着避免这种情况。

好吧，关于编程规范写的够多了；代码本身更有趣。尽情享受吧！