Google C++ Style Guide

**0 背景**

C++是Google大部分开源项目的主要编程语言之一。正如每一个C++程序员所知，C++有许多强大的特性，但是这种强大也不可避免的导致了它的复杂性，是代码更容易产生bug，难以阅读和维护。

本指南的目标就是通过详细地描述写C++代码时，哪些可以做，哪些不可以做，来驾驭其复杂性。这些规则保证代码易于管理，同时仍然允许高效地使用C++语言的特性。

Style，也可称为可读性，也可称为指导C++编程的约定。使用术语“style”有些用词不当，因为这些约定俗成不仅仅是源文件的格式化那么简单。

Google开发的大部分开源项目遵循本指南中的要求。

注意：本指南不是C++教程，我们假定读者非常熟悉C++。

**0.1 指导手册的目标**

**Why do we have this document**?

There are a few core goals that we believe this guide should serve. These are the fundamental whys that underlie all of the individual rules. By bringing these ideas to the fore, we hope to ground discussions and make it clearer to our broader community why the rules are in place and why particular decisions have been made. If you understand what goals each rule is serving, it should be clearer to everyone when a rule may be waived (some can be), and what sort of argument or alternative would be necessary to change a rule in the guide.

The goals of the style guide as we currently see them are as follows:

(1) Style rules should pull their weight

The benefit of a style rule must be large enough to justify asking all of our engineers to remember it. The benefit is measured relative to the codebase we would get without the rule, so a rule against a very harmful practice may still have a small benefit if people are unlikely to do it anyway. This principle mostly explains the rules we don’t have, rather than the rules we do: for example, goto contravenes many of the following principles, but is already vanishingly rare, so the Style Guide doesn’t discuss it.

(2) Optimize for the reader, not the writer

Our codebase (and most individual components submitted to it) is expected to continue for quite some time. As a result, more time will be spent reading most of our code than writing it. We explicitly choose to optimize for the experience of our average software engineer reading, maintaining, and debugging code in our codebase rather than ease when writing said code. "Leave a trace for the reader" is a particularly common sub-point of this principle: When something surprising or unusual is happening in a snippet of code (for example, transfer of pointer ownership), leaving textual hints for the reader at the point of use is valuable (std::unique\_ptr demonstrates the ownership transfer unambiguously at the call site).

(3) Be consistent with existing code

Using one style consistently through our codebase lets us focus on other (more important) issues. Consistency also allows for automation: tools that format your code or adjust your #includes only work properly when your code is consistent with the expectations of the tooling. In many cases, rules that are attributed to "Be Consistent" boil down to "Just pick one and stop worrying about it"; the potential value of allowing flexibility on these points is outweighed by the cost of having people argue over them.

Be consistent with the broader C++ community when appropriate

Consistency with the way other organizations use C++ has value for the same reasons as consistency within our code base. If a feature in the C++ standard solves a problem, or if some idiom is widely known and accepted, that's an argument for using it. However, sometimes standard features and idioms are flawed, or were just designed without our codebase's needs in mind. In those cases (as described below) it's appropriate to constrain or ban standard features. In some cases we prefer a homegrown or third-party library over a library defined in the C++ Standard, either out of perceived superiority or insufficient value to transition the codebase to the standard interface.

(4) Avoid surprising or dangerous constructs

C++ has features that are more surprising or dangerous than one might think at a glance. Some style guide restrictions are in place to prevent falling into these pitfalls. There is a high bar for style guide waivers on such restrictions, because waiving such rules often directly risks compromising program correctness.

Avoid constructs that our average C++ programmer would find tricky or hard to maintain

C++ has features that may not be generally appropriate because of the complexity they introduce to the code. In widely used code, it may be more acceptable to use trickier language constructs, because any benefits of more complex implementation are multiplied widely by usage, and the cost in understanding the complexity does not need to be paid again when working with new portions of the codebase. When in doubt, waivers to rules of this type can be sought by asking your project leads. This is specifically important for our codebase because code ownership and team membership changes over time: even if everyone that works with some piece of code currently understands it, such understanding is not guaranteed to hold a few years from now.

(5) Be mindful of our scale

With a codebase of 100+ million lines and thousands of engineers, some mistakes and simplifications for one engineer can become costly for many. For instance it's particularly important to avoid polluting the global namespace: name collisions across a codebase of hundreds of millions of lines are difficult to work with and hard to avoid if everyone puts things into the global namespace.

(6) 必要时让位于优化

性能优化有时是必要且恰当的，即使当和本文档中的其它原则是冲突的。

本文档的目的就是对合理限制提供最大的指导。一如既往，常识和品位应该盛行。有鉴于此，我们特别地引用了整个Google C++社区已有的习惯用法，不仅仅是你个人的喜好或你团队的那些约定。怀疑和不愿使用聪明的或不同寻常的构造：没有禁止并不等同于许可。自己拿主意，如果你不确定，那就毫不犹豫地问你项目领导，获得外部支援。

**1头文件**

通常，每一个.cc文件应该有一个相关的.h文件。但是有许多常用的例外，诸如单元测试和非常小的.cc文件，只包含一个main()函数。

正确地使用头文件，可以使你的代码在可读性，大小和性能上大为改观。

下面的规则可以避免在使用头文件时的各种陷阱。

**1.1 自包含头文件**

头文件应该是自包含的（只编译关于它们自己的内容）且以.h结束。那些意在包含的非头文件应该以.inc结束，但尽量少用。

所有的头文件应该是自包含的。用户和重构工具不应该为特定条件去包含某一个头文件。尤其是，头文件应该有头文件警卫（header guards）并且包含所有的它所需要的头文件。

在头文件中，优先使用模板和inline函数实现声明。这些构造的定义也应该被包含到每一个使用它们的.cc文件中，否则，程序就有可能会在构造配置阶段链接失败。如果声明和定义在不同的文件中，包含前者就会传递性地包含后者。不要再把定义分开包含在不同的头文件中（inl.h）；这种习惯在过去是常用的做法，但是现在不再被允许。

也有例外，为所有相关的模板参数集显式地实例化模板，或者其本身就是一个类的私有实现，那么它只能定义在实例化该模板的.cc文件里。

There are rare cases where a file designed to be included is not self-contained. These are typically intended to be included at unusual locations, such as the middle of another file. They might not use header guards, and might not include their prerequisites. Name such files with the .inc extension. Use sparingly, and prefer self-contained headers when possible.

**1.2 #define保护**

所有的头文件都应该使用#define保护，防止多重包含。命名格式如，

<PROJECT>\_<PATH>\_<FILE>\_H\_

为了保证唯一性，命名应该是基于项目源码树的全路径。例如，项目foo中的文件foo/src/bar/baz.h，应该使用下面的宏定义：

#ifndef FOO\_BAR\_BAZ\_H\_

#define FOO\_BAR\_BAZ\_H\_

...

#endif // FOO\_BAR\_BAZ\_H\_

**1.3 前置声明**

尽可能地避免使用前置声明，使用#include包含你所需要的头文件即可。

定义：

前置声明可以是类，函数或模板的声明，但没有相关的定义。

优点：

* 前置声明可以节省编译时间，因为#include强制编译器展开更多文件，且要处理更多的输入。
* 前置声明可以节省不必要的重编译时间。因为头文件中不相干的改动，#include会强制你的代码被重新编译多次。

缺点：

* + 前置声明能够隐藏依赖，当头文件发生改动时，允许用户代码跳过必要的重编译。
  + 前置声明可能会被库文件的后续改动破坏掉。函数和模板的前置声明阻止头文件开发者针对他们的API作出相应的改动。例如，扩展参数类型，使用默认值，或者迁移到新的命名命名空间。
  + 前置声明来源于命名空间std中的符号：其行为没定义。
* 使用前置声明还是使用#include，这很难抉择。用前置声明代替#include，有可能会暗暗地改变代码的含义。

// b.h:

struct B {};

struct D : B {};

且，

// good\_user.cc:

#include "b.h"

void f(B\*);

void f(void\*);

void test(D\* x) { f(x); } // calls f(B\*)

如果#include被B和D的前置声明取代，test()将会调用f(void\*)。

* + 前置声明了来自头文件的多个符号，比简单地#include头文件更冗长。
  + 仅仅为了前置声明而构建代码（比如使用指针成员代替对象成员）会使代码变得更慢更复杂。

结论：

* + 尽量避免前置声明那些定义在另一个项目中的实体
  + 当使用一个头文件中声明的函数时，总是#include那个头文件
  + 当使用模板类时，优先选择使用#include头文件

当#include头文件时，请查看《包含的路径和顺序》一章了解更多的规则。

**1.4 内联函数**

只有当当函数只有10行或者更少时才将其定义为内联函数。

定义：

一旦定义了内联函数，在某种程度上允许编译器将它们展开，而不是通过传统的函数调用机制调用它们。

优点：

只要内联函数够小，定义内联函数能够使目标代码更加高效。对于accessors存取器和mutators变值器，还有其它短的，高效的临界代码段，鼓励使用内联函数。

缺点：

事实上，过度使用内联，将会使程序更慢。依赖一个函数的大小，内联前后造成代码量的增加或者减少。内联一个非常小的accessor函数将会减少代码的大小，而内联一个非常大的函数可以动态的增加代码的大小。现代处理器因为更好的利用了指令缓存，所以小巧的代码能够更好地被运行。

结论：

一个合理的经验准则就是，不要内联超过10行的函数。谨慎对待析构函数，因为它们往往比看起来要长，因为隐含成员的调用和基类中析构函数的调用。

另一条有用的准则就是：不要内联循环或switch语句（除非，在正常情况下，循环和switch语句永远不会发生）。

还有重要的一点需要知道，即使函数被内联也不一定会被编译器编译成内联函数。例如，虚函数和递归调用通常不会被正常内联。通常情况下，递归函数不应该声明成内联函数。虚函数内联的一个主要原因就是：把它的函数体放在类定义中，既方便，也有助于当做文档记录其行为，例如，accessors存取器和mutators变值器。

**1.5 包含的路径和顺序**

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，其它库的.h头文件，项目相关的.h头文件。

所有的项目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..（父目录）。例如，google-awesome-project/src/base/logging.h应该这样被包含：

#include "base/logging.h"

又如，dir/foo.cc 或 dir/foo\_test.cc，他们主要的目的就是实现或测试dir2/foo2.h的功能，foo.cc文件中头文件的包含顺序如下：

* dir2/foo2.h.
* C system files.
* C++ system files.
* Other libraries' .h files.
* Your project's .h files.

使用这种优先的顺序，如果dir2/foo2.h遗漏了某些必要的包含时，dir/foo.cc 或 dir/foo\_test.cc的构建会被打断。因而，这条准则保证了首先展示给的是维护这些文件的人，而不是那些其它包的无关的人。

dir/foo.cc 和 dir2/foo2.h 大部分时候在同一目录下（例如，base/basictypes\_test.cc和base/basictypes.h），但是，也有可能在不同的目录下。

按照字母顺序分别对每种类型的头文件进行排序。注意，较老的代码可能不会遵循这条规则，要在方便的时候修正它们。

应该把你所有依赖的符号所在的头文件都包含进来，除非是前置声明那种特殊情况。如果你所依赖的符号在文件bar.h中，不要因为foo.h文件中已经包含了bar.h文件，而你已经包含了foo.h文件，就觉得不需要再包含bar.h文件了；请手动包含bar.h头文件，除非foo.h文件中明确说明它的目的就是提供给你bar.h文件中的符号。但是，任何在相关的头文件中存在的头文件包含不需要再被相关的.cc文件包含了（例如，foo.cc能够依赖foo.h里的头文件包含）。

例如，google-awesome-project/src/foo/internal/fooserver.cc文件中的头文件包含可能如下：

#include "foo/server/fooserver.h"

#include <sys/types.h>

#include <unistd.h>

#include <hash\_map>

#include <vector>

#include "base/basictypes.h"

#include "base/commandlineflags.h"

#include "foo/server/bar.h"

例外：

有时候，系统特殊代码需要条件编译。这类代码可以被放到其它包含后的条件包含中。当然了，保持你的系统特殊代码足够小且独立。例如：

#include "foo/public/fooserver.h"  
#include "base/port.h" // For LANG\_CXX11.  
#ifdef LANG\_CXX11  
#include <initializer\_list>  
#endif // LANG\_CXX11

2 作用域

2.1命名空间

一般情况下，我们都会把代码放在命名空间内，鲜有例外。命名空间的命名应该基于项目工程名称，最好包括路径，这样可以尽量保证其唯一性。不要使用指令（例如，禁止使用using namespace foo）。不要使用内联命名空间。对于不具名的命名空间，请看*不具名的命名空间和静态变量*。

定义：

命名空间把全局范围划分为不同的，命名的范围，这对防止全局范围内命名冲突是非常有用的。

优点：

命名空间提供了在大型程序中解决命名冲突的一种方法，同时还能允许大部分代码合理地使用缩写名称。

例如，如果两个不同的工程在全局范围内有一个类Foo，它们的符号可能会在编译时，或运行时发生冲突。如果每个工程都把自己的代码放在命名空间内，project1::Foo和project2::Foo是不同的符号，就不会冲突。每个工程中的代码可以继续引用Foo，而不用使用前缀。

内联命名空间自动地把它们的名字放进{}大括号括起来的范围内。考虑下面的代码段，

namespace X {

inline namespace Y {

void foo();

} // namespace Y

} // namespace X

表达式X::Y::foo()和X::foo()是可以替换的。内联命名空间主要解决跨版本的ABI兼容性。

缺点：

命名空间可能会令人困惑，因为它使得识别一个名称代表什么定义更加复杂了。

尤其是，内联函数更容易是人混淆，因为名称没有真实地限定在声明它们的地方的命名空间内。他们只有作为大型程序版本控制策略的一部分时才是有用的。

在某些场合下，必须重复地引用完全限定名称的符号，对于深度嵌套的命名空间，这会增加许多杂乱。

结论：

命名空间应该遵循下面的策略:

* 遵循*命令空间命名*中的规则；
* 向前面给出的几个例子一样，在命名空间的结束处，使用命名空间的名称进行注释；
* 除了文件包含，gflags定义或声明及其它命名空间的类的前置声明之外的源文件全部包含在命名空间的范围内；

// .h头文件

namespace mynamespace {  
// 所有的声明都在命名空间作用域范围内  
// 注意缩进格式.  
class MyClass {  
public:  
 ...  
 void Foo();  
};  
} // namespace mynamespace

// .cc 文件

namespace mynamespace {  
// 函数的定义也必须放在命名空间的作用域范围内

void MyClass::Foo() {  
 ...  
}  
} // namespace mynamespace

* 更为复杂的.cc文件，会有其它额外的内容，像标志符号或者使用声明。

#include "a.h"

DEFINE\_FLAG(bool, someflag, false, "dummy flag");

namespace a {

using ::foo::bar;

...code for a... // Code goes against the left margin.

} // namespace a

* 不要在命名空间std中声明任何东西，包括标准类库的前置声明。在命名空间std声明实体是未定义的行为，例如，不可移植性。为了声明来自标准库的实体，包含可利用的头文件就可以了。
* 你也不可以使用using指令，取得一个可用命名空间的所有名称。

// 禁止，这会污染命名空间缩短.cc文件长使用名称的名字。  
using namespace foo;

* 不要再头文件的命名空间范围内使用命名空间别名，除非在明确标记只是内部命名空间内。因为导入到头文件命名空间的任何东西都会成为公共API的一部分。

// 缩短.cc文件长使用名称的名字。  
namespace baz = ::foo::bar::baz;

// 缩短.h文件长使用名称的名字。

namespace librarian {

namespace impl { // 内部使用, 不是API的一部分.

namespace sidetable = ::pipeline\_diagnostics::sidetable;

} // namespace impl

inline void my\_inline\_function() {

// 函数（或方法）局部命名空间别名

namespace baz = ::foo::bar::baz;

...

}

} // namespace librarian

* 不要使用内联命名空间。

2.2 不具名的命名空间和静态变量

在.cc文件中定义的内容没有被其它文件引用的话，把这些定义放进不具名的命名空间中或将它们声明为static。但是不要在头文件中使用它们中的任何一个。

定义：

将所有的声明放入不具名的命名空间中，这些声明将会是内部链接；同样，将函数和变量声明为static，也将是内部链接。这意味着，无法从别的文件访问你所声明的任何内容。不同的文件中声明了相同名称的实体，它们将是完全独立，没有任何关系的。

结论：

在.cc文件中，对于所有不被其它地方引用的代码，鼓励使用内部链接。但是，不要在头文件中使用内部链接。

不具名命名空间最好使用下面的格式：

namespace {

...

} // namespace

2.3 非成员, 静态成员, 和全局函数

优先把非成员函数放入命名空间；尽量不要使用全局函数。优先使用命名空间组合函数，代替使用类组合函数。类的静态方法往往和类实例或者类的静态数据紧密相关。

优点：

在某些情况下，非成员和静态成员函数是有用的。把非成员函数放入命名空间例，避免污染全局命名空间。

缺点：

非成员和静态成员函数作为一个新类的成员会更有意义，尤其如果他们访问外部资源或者有显著的相关性。

结论：

有时候，声明一个没有绑定到类实例的函数是很有用的。这样的函数既可以是静态成员，也可以是非成员函数。非成员函数不应该依赖外部变量，且应该总是存在于命名空间内。而不是创建一个类，仅仅是为了组合不共享静态数据的静态成员函数，使用命名空间代替，对于头文件myproject/foo\_bar.h，例如

namespace myproject {

namespace foo\_bar {

void Function1();

void Function2();

} // namespace foo\_bar

} // namespace myproject

代替

namespace myproject {

class FooBar {

public:

static void Function1();

static void Function2();

};

} // namespace myproject

如果定义了一个非成员函数且只有.cc文件需要，使用内部链接限制它的作用范围。

2.4 局部变量

将函数变量尽可能置于最小作用域内，在声明时将其初始化。

C++允许在函数的任意位置声明变量。我们提倡尽可能在最小的作用范围内声明变量，尽可能离第一次使用的地方越近越好。这使得代码易于阅读，易于查找声明位置，变量类型和初始化值。尤其是，使用初始化声明代替声明+赋值的方式，如下所示：

int i;

i = f(); // 坏 – 初始化和声明分离

int j = g(); // 好 – 声明时初始化

std::vector<int> v;

v.push\_back(1); // 优先使用列表初始化

v.push\_back(2);

std::vector<int> v = {1, 2}; //好 – {}初始化

if, while,和for等语句需要的变量，应该在这些语句内部进行声明，也就是这些变量被限制在它们的范围之内（但这不是绝对的，下面会有讲解），例如：

while (const char\* p = strchr(str, '/')) str = p + 1;

警告：如果变量是一个对象，每次进入这个作用域，它的构造函数都会被调用，离开时，析构函数会被调用。

// 非有效实现

for (int i = 0; i < 1000000; ++i) {

Foo f; // 构造和析构函数将会被调用1000000 次

f.DoSomething(i);

}

如下声明可能更有效：

Foo f; // 构造和析构函数只会被调用一次

for (int i = 0; i < 1000000; ++i) {

f.DoSomething(i);

}

2.5 静态和全局变量

具有静态存储周期的class类型变量是被禁止的：因为构造函数和析构函数无法确定调用顺序，而导致很难查找bug。但是，如果变量是常量表达式的话，是被允许的：它们没有动态初始化或析构函数。

具有静态存储周期的对象，包括全局变量，静态变量，静态类成员变量，和函数的静态变量，必须是简单旧数据（Plain Old Data）：像int，char，float，或指针，再或者它们的数组/结构体。

类构造函数和静态变量初始化的顺序，在C++中只作了部分规定，并且这个顺序在每次构建时，都会发生变化，从而造成难以查找的bug。因此，除了禁止class类型的全局变量之外，我们也不允许使用函数的结果初始化非局部静态变量，除非是这样的函数，像getenv()，getpid()之类的，不依赖于任何其他全局变量的函数。但是，函数内部的静态POD变量可以使用函数的结果进行初始化，因为它的初始化顺序是定义好的，直到控制执行到这个声明的地方时才会发生初始化。

同样的，当程序终止时，全局变量和静态变量也会被销毁，不论程序是从main()函数正常return还是调用了exit()函数。被调用的析构函数的顺序正好与被调用构造函数的顺序相反。因为构造函数的顺序是不确定的，那么析构函数也是如此。例如，在程序的结束阶段，一个静态变量可能要被销毁，但与此同时，代码仍然在运行-或许是另一个线程-尝试访问这个静态变量且失败。再或者一个静态string变量的析构函数有可能比另一个包含那个string变量的变量的析构函数运行的早。

缓解析构函数问题的一个方法就是，调用quick\_exit()代替exit()函数终止程序。不同之处在于，quick\_exit()不调用析构函数且不会调用任何使用atexit()函数而注册的函数句柄。当程序通过quick\_exit()终止时，而你还想运行一个函数句柄（例如，刷新日志缓冲区中的日志），你可以使用at\_quick\_exit()注册它。（如果你需要在exit和quick\_exit都运行一个函数句柄，那么你需要在这两处注册它）

我们只允许静态变量包含POD数据的结果就是，std::vector使用C数组代替，或者字符串使用C风格的字符串const char []。

如果你确实需要一个class类型的静态或者全局变量，考虑初始化为指针（指针将永远不被释放）。需要注意的是，是原始指针，而不是智能指针，因为智能指针的析构函数也会存在我们极力避免的析构函数顺序问题。

**译者注：**

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

3 类

类是C++代码里的基本单元。自然地，我们会广泛地使用它。本节列出了在写一个类时，应该做什么，不应该做什么。

3.1 构造函数的职责

避免在构造函数里调用虚函数，也要避免在无法报出错误时，进行可能失败的初始化。

定义：

尽可能地在构造函数里执行初始化操作。

优点：

* 不需要担心类是否被初始化。
* 完全由构造函数初始化的对象可以是const类型，也很容易被标准容器和算法使用。

缺点：

如果调用了虚函数，调用就不能被派发到子类实现中。即使你的类当前没有被子类化，也可能会在将来的修改中隐含地引入这个问题，带来隐患。

构造函数没有简单的方法报告错误，不会导致程序崩溃或者禁止使用异常。

If the work fails, we now have an object whose initialization code failed, so it may be an unusual state requiring a bool IsValid() state checking mechanism (or similar) which is easy to forget to call.

You cannot take the address of a constructor, so whatever work is done in the constructor cannot easily be handed off to, for example, another thread.

结论：

Constructors should never call virtual functions. If appropriate for your code , terminating the program may be an appropriate error handling response. Otherwise, consider a factory function or Init() method. Avoid Init() methods on objects with no other states that affect which public methods may be called (semi-constructed objects of this form are particularly hard to work with correctly).

3.2 隐式转换

Do not define implicit conversions. Use the explicit keyword for conversion operators and single-argument constructors.

定义：

Implicit conversions allow an object of one type (called the source type) to be used where a different type (called the destination type) is expected, such as when passing an int argument to a function that takes a double parameter.

In addition to the implicit conversions defined by the language, users can define their own, by adding appropriate members to the class definition of the source or destination type. An implicit conversion in the source type is defined by a type conversion operator named after the destination type (e.g. operator bool()). An implicit conversion in the destination type is defined by a constructor that can take the source type as its only argument (or only argument with no default value).

The explicit keyword can be applied to a constructor or (since C++11) a conversion operator, to ensure that it can only be used when the destination type is explicit at the point of use, e.g. with a cast. This applies not only to implicit conversions, but to C++11's list initialization syntax:

class Foo {

explicit Foo(int x, double y);

...

};

void Func(Foo f);

Func({42, 3.14}); // Error

This kind of code isn't technically an implicit conversion, but the language treats it as one as far as explicit is concerned.

优点：

Implicit conversions can make a type more usable and expressive by eliminating the need to explicitly name a type when it's obvious.

Implicit conversions can be a simpler alternative to overloading.

List initialization syntax is a concise and expressive way of initializing objects.

缺点：

Implicit conversions can hide type-mismatch bugs, where the destination type does not match the user's expectation, or the user is unaware that any conversion will take place.

Implicit conversions can make code harder to read, particularly in the presence of overloading, by making it less obvious what code is actually getting called.

Constructors that take a single argument may accidentally be usable as implicit type conversions, even if they are not intended to do so.

When a single-argument constructor is not marked explicit, there's no reliable way to tell whether it's intended to define an implicit conversion, or the author simply forgot to mark it.

It's not always clear which type should provide the conversion, and if they both do, the code becomes ambiguous.

List initialization can suffer from the same problems if the destination type is implicit, particularly if the list has only a single element.

结论：

Type conversion operators, and constructors that are callable with a single argument, must be marked explicit in the class definition. As an exception, copy and move constructors should not be explicit, since they do not perform type conversion. Implicit conversions can sometimes be necessary and appropriate for types that are designed to transparently wrap other types. In that case, contact your project leads to request a waiver of this rule.

Constructors that cannot be called with a single argument should usually omit explicit. Constructors that take a single std::initializer\_list parameter should also omit explicit, in order to support copy-initialization (e.g. MyType m = {1, 2};).

3.3 拷贝和移动构造函数

Support copying and/or moving if these operations are clear and meaningful for your type. Otherwise, disable the implicitly generated special functions that perform copies and moves.

定义：

A copyable type allows its objects to be initialized or assigned from any other object of the same type, without changing the value of the source. For user-defined types, the copy behavior is defined by the copy constructor and the copy-assignment operator. string is an example of a copyable type.

A movable type is one that can be initialized and assigned from temporaries (all copyable types are therefore movable). std::unique\_ptr<int> is an example of a movable but not copyable type. For user-defined types, the move behavior is defined by the move constructor and the move-assignment operator.

The copy/move constructors can be implicitly invoked by the compiler in some situations, e.g. when passing objects by value.

优点：

Objects of copyable and movable types can be passed and returned by value, which makes APIs simpler, safer, and more general. Unlike when passing objects by pointer or reference, there's no risk of confusion over ownership, lifetime, mutability, and similar issues, and no need to specify them in the contract. It also prevents non-local interactions between the client and the implementation, which makes them easier to understand, maintain, and optimize by the compiler. Further, such objects can be used with generic APIs that require pass-by-value, such as most containers, and they allow for additional flexibility in e.g., type composition.

Copy/move constructors and assignment operators are usually easier to define correctly than alternatives like Clone(), CopyFrom() or Swap(), because they can be generated by the compiler, either implicitly or with = default. They are concise, and ensure that all data members are copied. Copy and move constructors are also generally more efficient, because they don't require heap allocation or separate initialization and assignment steps, and they're eligible for optimizations such as copy elision.

Move operations allow the implicit and efficient transfer of resources out of rvalue objects. This allows a plainer coding style in some cases.

缺点：

Some types do not need to be copyable, and providing copy operations for such types can be confusing, nonsensical, or outright incorrect. Types representing singleton objects (Registerer), objects tied to a specific scope (Cleanup), or closely coupled to object identity (Mutex) cannot be copied meaningfully. Copy operations for base class types that are to be used polymorphically are hazardous, because use of them can lead to object slicing. Defaulted or carelessly-implemented copy operations can be incorrect, and the resulting bugs can be confusing and difficult to diagnose.

Copy constructors are invoked implicitly, which makes the invocation easy to miss. This may cause confusion for programmers used to languages where pass-by-reference is conventional or mandatory. It may also encourage excessive copying, which can cause performance problems.

结论：

Provide the copy and move operations if their meaning is clear to a casual user and the copying/moving does not incur unexpected costs. If you define a copy or move constructor, define the corresponding assignment operator, and vice-versa. If your type is copyable, do not define move operations unless they are significantly more efficient than the corresponding copy operations. If your type is not copyable, but the correctness of a move is obvious to users of the type, you may make the type move-only by defining both of the move operations.

If your type provides copy operations, it is recommended that you design your class so that the default implementation of those operations is correct. Remember to review the correctness of any defaulted operations as you would any other code, and to document that your class is copyable and/or cheaply movable if that's an API guarantee.

class Foo {

public:

Foo(Foo&& other) : field\_(other.field) {}

// Bad, defines only move constructor, but not operator=.

private:

Field field\_;

};

Due to the risk of slicing, avoid providing an assignment operator or public copy/move constructor for a class that's intended to be derived from (and avoid deriving from a class with such members). If your base class needs to be copyable, provide a public virtual Clone() method, and a protected copy constructor that derived classes can use to implement it.

If you do not want to support copy/move operations on your type, explicitly disable them using = delete in the public: section:

// MyClass is neither copyable nor movable.

MyClass(const MyClass&) = delete;

MyClass& operator=(const MyClass&) = delete;

3.4 结构体和类对比

Use a struct only for passive objects that carry data; everything else is a class.

The struct and class keywords behave almost identically in C++. We add our own semantic meanings to each keyword, so you should use the appropriate keyword for the data-type you're defining.

structs should be used for passive objects that carry data, and may have associated constants, but lack any functionality other than access/setting the data members. The accessing/setting of fields is done by directly accessing the fields rather than through method invocations. Methods should not provide behavior but should only be used to set up the data members, e.g., constructor, destructor, Initialize(), Reset(), Validate().

If more functionality is required, a class is more appropriate. If in doubt, make it a class.

For consistency with STL, you can use struct instead of class for functors and traits.

Note that member variables in structs and classes have different naming rules.

3.5 继承

Composition is often more appropriate than inheritance. When using inheritance, make it public.

定义：

When a sub-class inherits from a base class, it includes the definitions of all the data and operations that the parent base class defines. In practice, inheritance is used in two major ways in C++: implementation inheritance, in which actual code is inherited by the child, and interface inheritance, in which only method names are inherited.

优点：

Implementation inheritance reduces code size by re-using the base class code as it specializes an existing type. Because inheritance is a compile-time declaration, you and the compiler can understand the operation and detect errors. Interface inheritance can be used to programmatically enforce that a class expose a particular API. Again, the compiler can detect errors, in this case, when a class does not define a necessary method of the API.

缺点：

For implementation inheritance, because the code implementing a sub-class is spread between the base and the sub-class, it can be more difficult to understand an implementation. The sub-class cannot override functions that are not virtual, so the sub-class cannot change implementation. The base class may also define some data members, so that specifies physical layout of the base class.

结论：

All inheritance should be public. If you want to do private inheritance, you should be including an instance of the base class as a member instead.

Do not overuse implementation inheritance. Composition is often more appropriate. Try to restrict use of inheritance to the "is-a" case: Bar subclasses Foo if it can reasonably be said that Bar "is a kind of" Foo.

Make your destructor virtual if necessary. If your class has virtual methods, its destructor should be virtual.

Limit the use of protected to those member functions that might need to be accessed from subclasses. Note that data members should be private.

Explicitly annotate overrides of virtual functions or virtual destructors with an override or (less frequently) final specifier. Older (pre-C++11) code will use the virtual keyword as an inferior alternative annotation. For clarity, use exactly one of override, final, or virtual when declaring an override. Rationale: A function or destructor marked override or final that is not an override of a base class virtual function will not compile, and this helps catch common errors. The specifiers serve as documentation; if no specifier is present, the reader has to check all ancestors of the class in question to determine if the function or destructor is virtual or not.

3.6 多重继承

Only very rarely is multiple implementation inheritance actually useful. We allow multiple inheritance only when at most one of the base classes has an implementation; all other base classes must be pure interface classes tagged with the Interface suffix.

定义：

Multiple inheritance allows a sub-class to have more than one base class. We distinguish between base classes that are pure interfaces and those that have an implementation.

优点：

Multiple implementation inheritance may let you re-use even more code than single inheritance (see Inheritance).

缺点：

Only very rarely is multiple implementation inheritance actually useful. When multiple implementation inheritance seems like the solution, you can usually find a different, more explicit, and cleaner solution.

结论：

Multiple inheritance is allowed only when all superclasses, with the possible exception of the first one, are pure interfaces. In order to ensure that they remain pure interfaces, they must end with the Interface suffix.

注意：

There is an exception to this rule on Windows.

3.7 接口

Classes that satisfy certain conditions are allowed, but not required, to end with an Interface suffix.

定义：

A class is a pure interface if it meets the following requirements:

* It has only public pure virtual ("= 0") methods and static methods (but see below for destructor).
* It may not have non-static data members.
* It need not have any constructors defined. If a constructor is provided, it must take no arguments and it must be protected.
* If it is a subclass, it may only be derived from classes that satisfy these conditions and are tagged with the Interface suffix.

An interface class can never be directly instantiated because of the pure virtual method(s) it declares. To make sure all implementations of the interface can be destroyed correctly, the interface must also declare a virtual destructor (in an exception to the first rule, this should not be pure). See Stroustrup, The C++ Programming Language, 3rd edition, section 12.4 for details.

优点：

Tagging a class with the Interface suffix lets others know that they must not add implemented methods or non static data members. This is particularly important in the case of multiple inheritance. Additionally, the interface concept is already well-understood by Java programmers.

缺点：

The Interface suffix lengthens the class name, which can make it harder to read and understand. Also, the interface property may be considered an implementation detail that shouldn't be exposed to clients.

结论：

A class may end with Interface only if it meets the above requirements. We do not require the converse, however: classes that meet the above requirements are not required to end with Interface.

3.8 运算符重载

Overload operators judiciously. Do not create user-defined literals.

定义：

C++ permits user code to declare overloaded versions of the built-in operators using the operator keyword, so long as one of the parameters is a user-defined type. The operator keyword also permits user code to define new kinds of literals using operator"", and to define type-conversion functions such as operator bool().

优点：

Operator overloading can make code more concise and intuitive by enabling user-defined types to behave the same as built-in types. Overloaded operators are the idiomatic names for certain operations (e.g. ==, <, =, and <<), and adhering to those conventions can make user-defined types more readable and enable them to interoperate with libraries that expect those names.

User-defined literals are a very concise notation for creating objects of user-defined types.

缺点：

Providing a correct, consistent, and unsurprising set of operator overloads requires some care, and failure to do so can lead to confusion and bugs.

Overuse of operators can lead to obfuscated code, particularly if the overloaded operator's semantics don't follow convention.

The hazards of function overloading apply just as much to operator overloading, if not more so.

Operator overloads can fool our intuition into thinking that expensive operations are cheap, built-in operations.

Finding the call sites for overloaded operators may require a search tool that's aware of C++ syntax, rather than e.g. grep.

If you get the argument type of an overloaded operator wrong, you may get a different overload rather than a compiler error. For example, foo < bar may do one thing, while &foo < &bar does something totally different.

Certain operator overloads are inherently hazardous. Overloading unary & can cause the same code to have different meanings depending on whether the overload declaration is visible. Overloads of &&, ||, and , (comma) cannot match the evaluation-order semantics of the built-in operators.

Operators are often defined outside the class, so there's a risk of different files introducing different definitions of the same operator. If both definitions are linked into the same binary, this results in undefined behavior, which can manifest as subtle run-time bugs.

User-defined literals allow the creation of new syntactic forms that are unfamiliar even to experienced C++ programmers.

结论：

Define overloaded operators only if their meaning is obvious, unsurprising, and consistent with the corresponding built-in operators. For example, use | as a bitwise- or logical-or, not as a shell-style pipe.

Define operators only on your own types. More precisely, define them in the same headers, .cc files, and namespaces as the types they operate on. That way, the operators are available wherever the type is, minimizing the risk of multiple definitions. If possible, avoid defining operators as templates, because they must satisfy this rule for any possible template arguments. If you define an operator, also define any related operators that make sense, and make sure they are defined consistently. For example, if you overload <, overload all the comparison operators, and make sure < and > never return true for the same arguments.

Prefer to define non-modifying binary operators as non-member functions. If a binary operator is defined as a class member, implicit conversions will apply to the right-hand argument, but not the left-hand one. It will confuse your users if a < b compiles but b < a doesn't.

Don't go out of your way to avoid defining operator overloads. For example, prefer to define ==, =, and <<, rather than Equals(), CopyFrom(), and PrintTo(). Conversely, don't define operator overloads just because other libraries expect them. For example, if your type doesn't have a natural ordering, but you want to store it in a std::set, use a custom comparator rather than overloading <.

Do not overload &&, ||, , (comma), or unary &. Do not overload operator"", i.e. do not introduce user-defined literals.

Type conversion operators are covered in the section on implicit conversions. The = operator is covered in the section on copy constructors. Overloading << for use with streams is covered in the section on streams. See also the rules on function overloading, which apply to operator overloading as well.

3.9 访问控制

Make data members private, unless they are static const (and follow the naming convention for constants). For technical reasons, we allow data members of a test fixture class to be protected when using Google Test).

3.10 声明顺序

Group similar declarations together, placing public parts earlier.

A class definition should usually start with a public: section, followed by protected:, then private:. Omit sections that would be empty.

Within each section, generally prefer grouping similar kinds of declarations together, and generally prefer the following order: types (including typedef, using, and nested structs and classes), constants, factory functions, constructors, assignment operators, destructor, all other methods, data members.

Do not put large method definitions inline in the class definition. Usually, only trivial or performance-critical, and very short, methods may be defined inline. See Inline Functions for more details.

3.11 函数

