Contents

Google C++ Style Guide 中文版
原始文档：http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml
基于版本：3.274

1 背景

C++ 是 Google 许多开源项目所使用的主要编程语言。正如每个 C++ 程序员所知，这门语言有许多强大的特性，但是随之而来的是其复杂性，这反过来又使代码更容易引入 Bug，难以阅读和维护。

这份指南的目标是通过详细描述在写 C++ 代码时什么可以做什么不可以做来管理这种复杂性。这些规则可以保持代码基本的可控性，同时又允许程序员高效地使用 C++ 的语言特性。

风格，或可读性，即我们管理 C++ 代码的约定。``风格" 这个术语有点不恰当，因为这些约定所覆盖的范围远远不只源码文件的格式。

保持代码易于管理的方法之一是加强一致性。任何程序员都可以快速理解另一个程序员的代码，这一点非常重要。保持统一的风格并遵循约定，意味着更容易使用 ``模式匹配" 来推断各种标识符的含义。创建通用性，必需的习惯用语和模式使得代码更容易理解。有时可能有充分的理由改变某些风格规则，但是尽管如此，我们还是遵守规则以保持一致性。

本指南声明的另一件事是 C++ 的特性臃肿。C++ 是门庞大的语言，拥有很多高级特性。有时我们限制甚至禁止使用某些特性。我们这样做来保持代码的简洁，以避免这些特性可能引起的各种错误和问题。本指南列出了这些特性并解释了为什么它们被限制使用。

Google 开发的开源工程均遵循本指南的要求。

注意本指南不是 C++ 教程：我们假设读者都熟悉 C++。

2 头文件

通常每个.cc 文件都要有个对应的.h 文件。也有一些常见的例外，如单元测试以及只包含 main() 函数的小.cc 文件。

正确使用头文件可以使用你代码的可读性、大小和性能都有很大的改观。

下面的规则会引导你规避使用头文件的各种陷阱。

2.1 #define 保护

所有的头文件都应该使用 **#define** 保护起来以阻止多次包含。符号名规则应该是 **<** 项目 **>_<** 路径 **>_<** 文件 **>_H_**。

为了保证唯一性，它们应该基于在工程源代码树中的全路径。如，工程 foo 中的文件 foo/src/bar/baz.h 应该有以下的保护：

```
#ifndef FOO_BAR_BAZ_H_
#define FOO_BAR_BAZ_H_

...

#endif  // FOO_BAR_BAZ_H_
```

2.2 前置声明

> 之前的版本这一节叫 ``头文件依赖"，对前置声明的好处和使用讲得很多。这一版指南调整了规则：明确禁止前置声明函数，不再要求使用前置声明，不鼓励前置声明模板。基本上开发中不用刻意考虑前置声明了。

可以对普通的类使用前置声明来避免不必要的 **#include**。

2.2.1 定义：

``前置声明" 是类、函数或模版的声明，但不带相关定义。通常可以使用客户代码中实际使用的符号前置声明来取代 #include 行。

2.2.2 优点：

- 不必要的头文件包含会强制编译器打开更多文件并处理更多输入。
- 同时也使用你的代码因为头文件的改变而更经常地被重编译。

2.2.3 缺点：

- 由于像模板、类型定义 (typedef)、默认参数以及 using 声明等特性，确定一个前置声明的正式形式可能比较困难。
- 可能难以确定在给定的代码片中使用前置声明还是完整包含头文件，特别是涉及降至类型转换时更是如此。极端情况下，使用前置声明代替 #include 会默默改变代码行为。
- 前置声明一个头文件中的多个符号比直接包含这个头文件更啰嗦。
- 函数或模板的前置声明阻止了头文件作者对 API 做一些兼容改变，如加宽参数类型，或给模板添加一个带默认值的参数。
- 前置声明 std 命名空间中的符号会引发未定义的行为
- 为了前置声明构造的代码（如使用指针成员代替对象成员）可能更慢更复杂。
- 前置声明的实际效率优势没有得到证明。

2.2.4 结论

- 当使用一个头文件中的函数时，包含它。
- 当使用一个类模板时，优先包含其头文件。

- 当使用普通类时，可以用前置声明，但是要小心那此前向声明能导致低效或错误的情况; 当不确定时，包含相应的头文件即可。

- 不是仅仅为了使用前置声明而使用指针代替数据成员。

对于那些提供了你所需要的声明或定义的文件，总是使用 #include; 不要依赖头文件中通过非直接引用的头文件传递来的符号。一个例外是 myfile.cc 文件可以依赖它对应的头文件 myfile.h 中的 #include 和前置声明。

2.3 内联函数

只内联小于 **10** 行代码的小函数。

2.3.1 定义：

你可以以一种方式来声明函数，以允许编译器将它们就地展开，而不是通过普通函数调用机制来调用。

2.3.2 优点：

只要函数足够小，将其内联可以产生更高效的目标代码。对于存取函数和一些性能关键的小函数，可以放心使用内联。

2.3.3 缺点：

过度使用内联会导致程序变慢。内联可能使代码变大或变小，这取决于函数大小。内联一个很小的存取函数通常会减小代码大小，但是内联一个很大的函数则可能使代码体积暴增。利用指令缓存，在现代处理器上小代码通常跑得更快。

2.3.4 结论：

一个合理的经验准则是：不要内联超过 10 行的函数。要警惕析构函数，由于隐含成员和基类析构函数的调用，它们往往比表面上看起来要长。

另一个有用的经验法则：内联那些含有循环或 switch 语句的函数通常是不划算的（除非，这循环或 switch 语句通常都不会执行到）。

还有一点很重要，即使函数被声明为内联的，它们也不一定总是会被内联；比如，虚函数和递归函数就不能正常内联。通常递归函数不应该被内联。内联一个虚函数的主要原因是要将其定义放在类中，这是为了方便或是文档化其自身的行为，比如存取函数。

2.4 -inl.h 文件

当需要时，你可以使用后缀名为 **-inl.h** 的文件来定义复杂的内联函数。

内联函数的定义要放在一个头文件中，这样编译器才能函数在调用处知道其定义。然而，实现代码应该是属于.cc 文件的，除非有明显的可读性和性能优势，否则我们并不想在.h 文件放太多的实现代码。

如果一个内联函数很短，里面只有很少的一点逻辑，你应该把代码放在.h 文件中。比如存取函数就应该放在类定义中。出于对实现者和调用者的方便，更复杂的内联函数也可以放在.h 文件中，如果这使得.h 文件变得过于笨重，你也可以把其代码放到一个单独的 -inl.h 文件中。把实现和类定义分开，当需要时仍经允许包含实现头文件。

-inl.h 文件的另一个用处是定义函数模板。这可以使你的模板定义更易阅读。

不要忘记 -inl.h 文件也其它的头文件一样需要 #define 保护.

2.5 函数参数顺序

当定义一个函数时，参数顺序为：输入，然后是输出。

C/C++ 函数的参数或是输入或是输出或两者皆是。输入参数通常是值或常量引用，而输出或输入/输出参数是非常量指针。当安排参数顺序时，把所有的输入参数放到输出参数前面。尤其不要因为一个参数是新添加的就把它放在最后面；新的输入参数也要放在输出参数前面。

这不是硬性规定。既是输入又是输出的参数（通常是类或结构体）把事情变得复杂，同时，为了和相关函数保持我们一惯主张的一致性，你有时可以有所变通。

2.6 名称以及包含顺序

使用标准的头文件包含顺序增强可读性，并避免隐藏依赖：**C** 库，**C++** 库，其它库头文件，本项目库头文件。

所有本工程的头文件应该按源代码目录树结构排列，避免使用 UNIX 的特殊缩写.（当前目录）或..（父目录）。比如，google-awesome-project/src/base/logging.h 应该这样被包含：

```
#include "base/logging.h"
```

在 dir/foo.cc 或 dir/foo_test.cc 中，其主要目的是实现或测试 dir2/foo2.h 中的声明的东西, 你的包含次序应该是这样的：

1. dir2/foo2.h (优先位置 --- 后面会详述).
2. C 系统文件。
3. C++ 系统文件。
4. 其它库的.h 文件。
5. 本工程内的.h 文件。

按这种顺序，如果 dir/foo2.h 遗漏了必要的头文件，dir/foo.cc 或 dir/foo_test.cc 的编译就会失败。这样，这条规则就保证了编译首先在使用这些文件的人面前失败，而不是使用其它包的无辜的人。

dir/foo.cc 和 dir2/foo2.h 通常在同一目录中（比如 base/basictypes_test.cc 和 base/basictypes.h），但也可以在不同的目录中。

在上面的每一块中，按字母顺序排列是个好主意。注意旧代码可以不符合这条规则，但应该在方便的时候修改它。

比如，google-awesome-project/src/foo/internal/fooserver.cc 中的包含次序应看起来像这样：

```
#include "foo/public/fooserver.h"  // Preferred location.

#include <sys/types.h>
#include <unistd.h>
#include <hash_map>
#include <vector>

#include "base/basictypes.h"
#include "base/commandlineflags.h"
#include "foo/public/bar.h"
```

例外：有时，系统相关的代码需要条件包含。这样的代码可以把条件包含放到其它的包含之后。当然，要保持系统相关代码短小并局部化。如：

```
#include "foo/public/fooserver.h"

#include "base/port.h"  // For LANG_CXX11.

#ifdef LANG_CXX11
#include <initializer_list>
#endif  // LANG_CXX11
```

3 作用域

3.1 命名空间

.cc 文件中鼓励使用匿名命名空间。使用具名命名空间时，选择基于工程的名字，尽可能使用其路径。不要使用 **using** 指令。不要使用内联命名空间

3.1.1 定义：

命名空间把全局作用域划分为独立的具名作用域，以此可以有效防止全局作用域中的命名冲突。

3.1.2 优点：

命名空间在类（可嵌套的）外又提供了一层命名轴线（也是可嵌套的）。

比如，如果两个工程的全局作用域中都有一个名为 Foo 的类，这在编译或进行时就会造成冲突。如果每个工程都把它们的代码放在一个命名空间中，这样 project1::Foo 和 project2:Foo 就是两个不同的符号，自然没有冲突。

内联命名空间自动将其名称放置到封闭作用域中。举个例子，考虑下面的代码片：

```
namespace X {
inline namespace Y {
  void foo();
}
}
```

表达式 `X::Y::foo()` 和 `X::foo()` 是可替换的。内联命名空间主要是为了跨版本的 ABI 兼容性。

3.1.3 缺点：

正是由于和类一样提供了额外的（都是可嵌套的）名字轴线，命名空间具有一些迷惑性。

特别是内联命名空间，更可能混淆，因为名字在其声明的命名空间内不是强制的。是在作为更大版本策略的一部分时才有用。

在头文件中使用匿名空间很容易违背 C++ 的唯一定义原则（ODR，One Definition Rule）。

3.1.4 结论：

要根据以下策略使用命名空间。要像示例中那样在命名空间结束处使用注释。

3.1.4.1 匿名空间

- 匿名空间在.cc 中是允许的，甚至是被鼓励的，以此可以避免命名冲突：

```
namespace {                                  // 这个一个.cc 文件中的代码

// 命令空间的内容不缩进
enum { kUnused, kEOF, kError };          // Commonly used tokens.
bool AtEof() { return pos_ == kEOF; }    // Uses our namespace's EOF.

}  // namespace
```

尽管如此，特定类相关的文件作用域声明，可以在类中被声明为类型、静态成员或静态成员函数，而不是某个匿名空间的成员。

- 不要在.h 文件中使用匿名空间。

3.1.4.2 具名空间 具名空间应该像下面这样使用：

- 使用命名空间把除头文件包含，gflags 定义/声明以及类的前置声明以外的整个代码包装起来，以别于其它命令空间：

```
// In the .h file
namespace mynamespace {

// 所有声明都置于命名空间中。
// 注意没有缩进
class MyClass {
 public:
  ...
  void Foo();
};

}  // namespace mynamespace

// In the .cc file
namespace mynamespace {

// 函数定义在命名空间作用域中
void MyClass::Foo() {
  ...
}

}  // namespace mynamespace
```

通常.cc 文件会有更多复杂细节，包括引用其它命名空间中的类。

```
#include "a.h"

DEFINE_bool(someflag, false, "dummy flag");

class C;  // 全局命名空间中类 C 的前置声明
namespace a { class A; }  // a::A 的前置声明

namespace b {

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

}  // namespace b
```

- 不要在 std 空间中声明任何东西，甚至不要前置声明标准库的类。在 std 空间中声明东西会产生不确定的行为，比如不可移植。要声明标准库中的东西，包含相应的头文件就好了。

- 不要使用 using 指令来引入一个命名空间中的所有名字。

```
// 禁止!!! -- 这会污染命令空间
using namespace foo;
```

- 在.cc 文件中的任何地方，.h 文件中的函数、方法以及类中都可以使用 using 声明

```
// 在.cc 文件这是允许的
// .h 文件中，必须是在函数、方法或类内部
using ::foo::bar;
```

- 在.cc 文件任何地方，在包装整个.h 文件的命名空间内的任何地方，以及在函数和方法中，都允许使用命名空间别名。

```
// .cc 文件中为常用的起短名
namespace fbz = ::foo::bar::baz;

// .h 文件中为常用的起短名
namespace librarian {
// 下面这些别名在所有包含此头文件的文件中均可见（当然是在 librarian 空间中）:
// 所以一个工程中别名的选择应该保持一致
namespace pd_s = ::pipeline_diagnostics::sidetable;

inline void my_inline_function() {
  // 只在一个函数或方法作用域中的命名空间别名.
  namespace fbz = ::foo::bar::baz;
  ...
}
}  // namespace librarian
```

注意，头文件中的别名在任何包含它的文件中均可见，所以公共头文件（那些要在工程外使用的）和它们传递包含的头文件中都要避免使用别名，这是保证公共 API 尽可能小的举措之一。

- 不要使用内联命名空间

3.2 嵌套类

当嵌套类是接口的一部分时，尽管你可以使用，但还是应该考虑使用命名空间来将其声明从全局作用域中分开。

3.2.1 定义：

一个类中可以定义另一类; 这个嵌套类也被称为成员类。

```
class Foo {

 private:
  // Bar is a member class, nested within Foo.
  class Bar {
    ...
  };

};
```

3.2.2 优点：

当嵌套类（或成员类）只被外围类中使用时是非常有用的; 把它作为成员类放到外围类的作用域中，而不用类名去污染外层作用域。嵌套类可以在外围类中前置声明，然后在.cc 文件中定义，这样可以避免在外围类的声明中定义嵌套类，因为嵌套类的定义往往只与实现有关。

3.2.3 缺点：

嵌套类只有在外围类的定义内部才能前置声明。因此，任何使用了 Foo::Bar* 指针的头文件都要包含 Foo 类完整的声明。

3.2.4 结论：

除非嵌套类是接口的一部分，如持有一些方法的选项，否则不要将其定义为公有的。

3.3 非成员函数，静态成员函数，和全局函数

尽量使用命名空间中的非成员函数或静态成员，而少用全局函数。

3.3.1 优点：

非成员函数和静态成员在某些情况下会非常有用。将非成员函数放在一个命名空间中以避免污染全局命名空间。

3.3.2 缺点：

非成员函数和静态成员通常更应该成为一个新类的成员，特别是当它们访问外部资源或有显著的依赖关系是，更应如此。

3.3.3 结论：

有时候定义一个与类实例无关的函数很有用，甚至是必要的。这样的函数可以是一个静态成员或非成员函数。非成员函数不应该依赖外部变量，并且总是尽量放在一个命名空间中。相比单纯创建一个类来组织静态成员函数而不共享任何数据，使用命名空间更合适。

定义在同一编译单元里的函数在被从其它编译单元直接调用时，会引入不必要的耦合和运行时依赖，静态成员函数尤是如此。考虑将这些函数提取到一个新类，或放置于一个独立库的命名空间中。

如果你要必须定义一个非成员函数并且它只在其所在的.cc 文件中使用，使用匿名命名空间或 static 关键字（如 `static int Foo(){...}`）来限制其作用域。

3.4 局部变量

尽量将函数变量放在一个较小的作用域内，并在声明时进行初始化。

C++ 允许你在函数的任何位置声明变量。我们鼓励你在尽可能小的作用域内声明，离第一次使用越进越好。这使得读者更容易看到变量的声明、定义以及初始值。特别提醒，应该初始化而非声明再赋值，如：

```
int i;
i = f(); // 不好 -- 初始化和声明分开。

int j = g(); // 好 -- 声明时初始化。

vector<int> v;
v.push_back(1);  // 应优先使用括号初始化。
v.push_back(2);

vector<int> v = {1, 2};  // 好 -- 变量 v 声明时初始化。
```

注意 gcc 正确实现了 `for (int i = 0; i < 10; ++i)`(i 的作用域仅为此 for 循环)，所以在同一作用域的其它 for 循环中可以重复使用 i。将声明作用域限定于在 while 和 if 语句中也是正确的，如：

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

有一点需要注意：如果变量是一个对象，它的构造函数会在每一次进入作用域创建对象时被调用，它的析构函数也会在每次离开作用域时被调用。

```
// 无效率的实现：
for (int i = 0; i < 1000000; ++i) {
  Foo f;  // Foo 的构造函数和析构函数分别被调用了 1000000 次。
  f.DoSomething(i);
}
```

这时在循环之外定义循环中使用的变量要高效的多：

```
Foo f;  // Foo 的构造函数和析构函数只分别被调用了 1 次。
for (int i = 0; i < 1000000; ++i) {
  f.DoSomething(i);
}
```

3.5 静态和全局变量

禁止使用类静态和全局变量：由于构造和析构顺序的不确定定，它们会引入难以查找的 **BUG**。然而，如果是 **`constexpr`**，则这些变量是允许的：它们没有动态初始化或析构。

有静态存储周期的变量，包括全局变量，静态变量，静态类成员变量以及函数的静态变量，必须是原生数据类型（POD：Plain Old Data）：只能是整数类型，字符型，浮点型，或指针，或 POD 类型数据的数组或结构体。

静态变量的初始化顺序在 C++ 中只有部分定义，甚至在每次构建时还会变化，这会引起难以查找的 BUG。因此在禁止类全局变量的同时，我们也不允许用一个函数的返回值初始化一个静态的 POD 变量，除非这个函数（如 getenv()，或 getpid()）本身不依赖于任何其它全局变量。

同样，全局和静态变量在程序结束时被销毁，无论是以 `main()` 返回还是调用 `exit()` 结束。析构函数的调用顺序被定义为和构造函数调用顺序相反。由于构造顺序不确定，析构顺序自然也是如此。例如，程序结束时一个静态变量可能已经被销毁了，但是程序还在跑 -- 可能在另一个线程中 -- 这时试图访问此变量就会失败。又或是一个静态 `string` 变量的析构函数可以会比另一个持有此变量引用的类先析构。

减轻这种析构问题的一个方法是调用 `quick_exit()` 来结束程序，而不是 `exit()`。不同点就在于 `quick_exit()` 不会调用析构函数，也不会调用任何 `atexit()` 注册的处理器。如果你需要在程序以 `quick_exit()` 结束时运行一个处理器（如刷新日志），你可以使用 `at_quick_exit()` 来注册它。（如果你有一个想在 `exit()` 和 `quick_exit()` 时都运行的处理器，需要两处均注册。）

综上所述，我们只允许 POD 类型的静态变量。这条规则完全禁止了 `vector`（可以使用 C 数组代替）或 `string`（可以使用 `const char []` 代替）。

如果你需要一个 class 类型的静态或全局变量，可以在 `main()` 函数或 `pthread_once()` 函数中考虑初始化一个指针（永不释放）。注意此处的指针必须是原生指针，而不能是智能指针，因为智能指针的析构函数也有顺序问题，这正是我们极力避免的。

4 类

类是 C++ 代码的基本单元，我们自然会广泛使用。这一节列出了我们在写一个类时应该遵循的主要规则。

4.1 在构造函数中做事

在构造函数中要避免执行复杂的初始化（尤其是那些可能失败或需要调用虚方法的初始化）。

4.1.1 定义：

在构造函数中可以进行初始化操作。

4.1.2 优点：

打字方便。不需要纠结类是否已经初始化。

4.1.3 缺点：

在构造函数中进行实质操作的问题在于：

- 构造函数无法报告错误，且不能使用异常。
- 一旦失败，我们就持有了一个初始化不完全的对象，这可能是一个不确定的状态。
- 如果调用了虚函数，这些调用不会被分派给子类的实现。即使你的类现在没有子类，未来的改动也可能会默默引入这个问题，这会引起很大的混乱。
- 如果有人创建了这个类型一个全局变量（这违背了上面的规则，但总有人会这么做），构造函数会在 `main()` 函数之前调用，这时的一些隐含的假设可能并不成立。如，gflags 此时尚未初始化。

4.1.4 结论：

构造函数永远不要调用虚函数，也不要抛出非致命的错误。如果你的对象需要实质初始化，可以考虑使用一个工厂函数或 `Init()` 方法。

4.2 初始化

如果你的类中定义了成员变量，你就必须了每个成员变量提供一个类内部的初始化函数或写一个构造函数（可以是一个默认构造函数）。如果没有声明任何构造函数，编译器会为你生成一个默认构造函数，它可能会使某些字段未初始化或初始化成不合适的值。

4.2.1 定义：

当我们不带参数 `new` 一个类对象，默认构造函数会被调用。当调用 `new[]`（创建数组）时也总会被调用。类内部成员初始化的意思是使用像 `int count_=17;` 或 `string name_{"abc"};` 这样的结构声明一个成员变量，而不是仅仅声明 `int count_;` 或 `string name_;`。

4.2.2 优点：

如果没有提供初始化器，就用一个用户定义的默认构造函数来初始化一个对象。这可以保证对象一构造完就处于一个合法并可用的状态；也可以保证一个对象开始是以一个明显 ``不可能" 的状态创建的，以便于调试。

类内部初始化保证了一个成员变量会正确地初始化，而无需在多个构造函数中复制初始化代码。这在你添加了一个新成员变量，在一个构造函数中初始化之，但忘了在另一个构造函数中放置初始化代码时会减少 bug。

4.2.3 缺点：

显式定义默认构造函数对你，代码作者，而言是额外的工作。

如果一个变量在声明时进行了初始化，在一个构造录函数中也进行了初始化，类内部成员初始化就可能产生混淆，因为构造函数中的值会覆盖声明中的值。

4.2.4 结论：

如果一个类中定义了成员变量又没有其它的构造函数，就必须定义一个默认构造函数（没有参数的构造函数）。将类以一种内部状态一致且有效的方式初始化无疑是更可取的。

对简单初始化，特别是当一个成员变量在多于一个构造函数中必须以相同方式初始化时，就使用类内部成员初始化。

如果你的类定义了成员变量，又没有类内部初始化，且如果没有其它构造函数，你就必须定义一个默认构造函数（没有参数的构造函数）。将类以一种内部状态一致且合法的方式初始化无疑是更可取的。

这么做的原因是，如果没有其它构造函数又没有定义默认构造函数，编译器就会帮你生成一个，而编译生成的这个构造函数未必能合理初始化你的对象。

如果你的类继承自其它类，你又没有添加新的成员变量，那就不必非要有一个默认构造函数。

4.3 显式构造函数

对只有一个参数的构造函数使用 **C++** 的 **`explicit`** 关键字。

4.3.1 定义：

通常一个只有一个参数的构造函数，可以被用作隐式转换。例如，你定义了 `Foo::Foo(string name)`，然后传递一个 string 给一个以 `Foo` 为参数的函数，此时这个构造函数会被调用，把 `string` 转换为 `Foo`，然后将它传递给函数。这可能带来方便，但是当转换以及新对象的创建不是你预期的时候，这就成了麻烦之源。用 `explicit` 来声明一个构造函数，可以阻止它被隐式调用而形成的转换。

4.3.2 优点：

避免不合适的转换。

4.3.3 缺点：

没有。

4.3.4 结论：

我们要求所有单参数的构造函数都是显式的。在类的单参数构造定义前都加上 `explicit`：`explicit Foo(string name);`。

拷贝构造函数是一个例外，在为数不多的被允许使用的情况下，它都不应是显式的。作为其它类的透明包装器的类也是个例外。这些例外都应该用注释清楚标注出来。

最后，只接收一个初始化列表的构造函数应该是非显式的。这是为了允许用大括号初始化列表赋值的方式构造类（如 `MyType m = {1, 2}`）。

initializer_list 是 C++0x 引入的特性

4.4 拷贝构造函数

只在必要的时候才提供拷贝构造函数和赋值操作符。否则，应使用 **DISALLOW_COPY_AND_ASSIGN** 宏来禁止它们。

4.4.1 定义：

拷贝构造函数和赋值操作符用来构造一个对象的拷贝。在某些情况下编译器会隐式调用拷贝构造函数，如，在按值传递对象时。

4.4.2 优点：

拷贝构造函数使拷贝对象变得简单。STL 容器要求所有的内容都要是可拷贝且可赋值的。拷贝构造函数比 `CopyFrom()` 式的方法更高效，因为把构造和拷贝结合在一起，编译器在某些情况下可以省略它们，并且更容易避免堆内存分配。

4.4.3 缺点：

隐式对象拷贝是 C++ 中许多 bug 和性能问题的根源。同时也降低了可读性，因为相比引用传递，跟踪值传递的对象变得困难，也难以确定哪些改变会反映出去。

4.4.4 结论：

只有少数的类需要可拷贝。大多数类都不应该有拷贝构造函数和赋值操作符。大多数情况下，一个指针或引用可以代替值拷贝，性能还更好。如，你可以传递给函数一个引用或指针而非一个值，也可以在 STL 容器中保存指针而非对象。

如果你的类需要可拷贝，优先使用一个拷贝方法，如 `Copyfrom()` 或 `Clone()`，而不是使用拷贝构造函数，因为这些方法不会被隐式调用。如果拷贝函数不满足你的要求（如性能原因，或你的类需要在 STL 容器中存值），那就 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，就必须显式禁止它们。可以在 class 的 private 区为构造函数和赋值操作符添加假的声明，但不提供相关定义（这样所有使用它们的企图都会引发一个链接错误）。

方便起见，一个 DISALLOW_COPY_AND_ASSIGN 宏可以这样使用:

```
// 一个禁止拷贝构造函数和赋值操作符的宏
// 它应该被用在类的 private 声明区
#define DISALLOW_COPY_AND_ASSIGN(TypeName) \
  TypeName(const TypeName&);               \
  void operator=(const TypeName&)
```

然后，在类 Foo 中：

```
class Foo {
 public:
  Foo(int f);
  ~Foo();

 private:
  DISALLOW_COPY_AND_ASSIGN(Foo);
};
```

4.5 委托和继承构造函数

当有助于降低代码重复率时，可以使用委托和继承构造函数。

4.5.1 定义

委托和继承构造函数是 C++11 中引入的两个不同特性，意在降低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接着类的另一个函数继续工作，使用了初始化列表语法的一个特殊变体。如：

```
X::X(const string& name) : name_(name) {
  ...
}

X::X() : X("") { }
```

继承构造函数一个派生类可以直接使用基类的构造函数，就像基类的其它成员函数一样，无需重新声明了。这在基类有多个构造函数时尤其有用。如：

```
class Base {
public:
  Base();
  Base(int n);
  Base(const string& s);
  ...
};

class Derived : public Base {
public:
  using Base::Base;  // Base's constructors are redeclared here.
};
```

这在 `Derived` 的构造函数除了调用 `Base` 的构造函数没有其它操作时特别有用。

4.5.2 优点：

委托和继承构造函数减少了冗余和样板，可以增加可读性。

委托构造对 Java 程序员来说很熟悉。

4.5.3 缺点：

使用帮助函数也可以近似模拟委托构造函数。

当派生类引入新的成员变量时继承构造函数可能引起混淆，因为基类的构造函数不知道它们。

4.5.4 结论

当可以降低冗余提高可读性时才使用委托和继承构造函数。但当派生类中有新成员变量时要小心继承构造函数。如果你对派生类的成员变量使用类内部初始化，继承构造函数就仍然是合适的。

4.6 结构体 VS 类

只持有数据的被动对象才能作为结构体; 其它的都应该用类。

C++ 中 struct 关键字和 class 关键字的行为几乎一样。我们给每个关键字添加了自己的语义，所以你应该为你定义的数据类型使用合适的关键字。

结构体只能用在只持有数据的被动对象上，并且可以有相关的常量，但是除了存取数据成员外没有任何其它功能。对数据成员的存取都是通过直接访问而不是调用方法。结构体的方法都不是私有的，但是仅限于操作数据成员，如，构造函数，析构函数，`Initialize()`，`Reset()`，`Validate()`。

如果需要更多功能，类更合适。如果不确定，就用类。

为了与 STL 保持一致，对于 functors 和 traits 你可以使用结构体代替类。

注意结构体和类的成员变量有不同的命名规则。

4.7 继承

组合通常都比继承更合适。当使用继承时，只能用公有继承。

4.7.1 定义：

当一个子类继承一个基类时，他就包含了父类中定义的所有数据和操作。实践中，继承在 C++ 中有两种主要用法：实现继承，实际代码都被子类继承; 还有接口继承，只有方法名被继承。

4.7.2 优点：

通过原样复用基类代码，实现继承减少了代码量。由于继承是编译期声明的，程序员和编译器都可以理解操作并检查错误。接口继承可用以强制暴露特定的 API。在这种情况下，当一个类没有定义必要的 API 方法时，编译器也可以检查到。

4.7.3 缺点：

对于实现继承，实现子类的代码在基类和子类间传播，这使得理解一个实现更为困难。子类不能重写非虚函数，所以子类也不能改变其实现。基类可能也定义了一些数据成员，所以还需要指明基类的物理布局。

4.7.4 结论：

所有的继承都应该是公有的。如果你想要一个私有继承，那么你应该把基类的实例作为成员包含进来。

不要过度使用实现继承。使用组合往往更合适。试着把继承的使用限制在 ``is-a'' 的情况：只有 `Bar` 确实是一种 `Foo` 时，`Bar` 才能作为 `Foo` 的子类。

尽可能使用虚析构函数。如果类中有虚方法，那么析构函数就一定要是虚方法。

仅对于那些可能在子类中访问的成员函数才使用 protected。注意数据成员都应该是私有的。

重定义一个继承的虚函数，要在声明中显式使用 virtual 关键字。原因是：如果省略 virtaul，为了搞清楚一个函数是不是虚函数，读者就需要检查所有的父类。

4.8 多重继承

多重继承只有极少的情况下有用。只有在最多一个基类有实现，其它的基类都是被标以 **Interface** 后缀的纯接口时，我们才允许使用多重继承。

4.8.1 定义：

多重继承允许一个子类可以有多个基类。我们需要把纯接口的基类和有实现的基类加以区别。

4.8.2 优点：

多重继承可能让你比单继承（见继承）复用更多的代码。

4.8.3 缺点：

多重继承真正有用的情况极少。当多重继承看似适用时，你往往可以发现另一种更明确，更干净的解决方案。

4.8.4 结论：

多重继承只有在除第一个基类外都是纯接口时才允许使用。为了确保它们是纯接口，必须以 Interface 为后缀。

注意：该规则在 Windows 下有个特例。

4.9 接口

满足下面条件的类允许使用 **Interface** 作为后缀，但不是强制的。

4.9.1 定义：

满足下面要求的类被称为纯接口：

- 只有公共的纯虚方法 (``= 0'') 以及静态方法（需要下文提到的析构函数）。
- 没有非静态数据成员。
- 不需要定义任何构造函数。如果提供了构造函数，那么一定是无参并且 protected 的。
- 如果是一个子类，只能从满足这些条件的被标以 Interface 后缀的类继承。

由于声明了纯虚方法，接口类不能被直接实例化。为了确保接口的所有实现都能正确销毁，接口类必须提供一个虚析构函数（必须不是纯虚的，尽管这违反了第一个条件）。关于这一点，在 Stroustrup 的《The C++ Programming Language, 3rd Edition》中的第 12.4 节有详细描述。

4.9.2 优点：

一个类被标以 Interface 后缀可以提醒其它人一定不能给它添加实现的方法或非静态数据成员。这个在多重继承中尤为重要。另外，对于 Java 程序员，接口的概念已经深入人心。

4.9.3 缺点：

Interface 后缀使类名变长，可能会给阅读和理解带来不便。同时，接口属性作为一种实现细节不应暴露给使用者。

4.9.4 结论：

只有当一个类满足上述条件时才可以使用 Interface 后缀。反过来我们不做要求：满足上述条件的类不强制以 Interface 为后缀。

4.10 运算符重载

除了极少数特殊情况，不要重载运算符。不要创建用户自定义的字面量。

4.10.1 定义：

一个类可以定义作用于此类的 + 和/等运算符，使其可以像内建运算符一样使用。一个 `operator""` 的重载允许使用内建字面量语法创建类的对象。

4.10.2 优点：

运算符重载可以使代码看上去更直观，因为一个类的行为会和内建类型（如 int）一致。重载的运算符比形如 Equals() 和 Add() 的函数名更出彩。

为了使有些模板函数正确工作，你可能需要定义操作符。

用户定义的字面量是创建户自定义类型对象的一种非常简洁的方法。

4.10.3 缺点：

尽管重载运算符使代码更直观，它却有几个缺点：

- 会混淆视听，让我们误以为一些耗时的操作和内建操作一样轻巧。
- 查找重载运算符的调用点变得困难得多。查找 Equals() 要比查找相关的 == 调用容易得多。
- 有些运算符也操作指针，重载它们很容易引入 bug。Foo + 4 做一件事，&Foo + 4 可能与之完全不同。编译器对这两种情况都不会提示，这使得调试异常困难。
- 用户自定义字面量允许创建即使是有经验的 C++ 程序员也不熟悉的新语法形式

重载还有想像不到的副作用。如，一个类重载了一元操作符 operator&，那么它就不能安全的前置声明了。

4.10.4 结论：

通常都不要重载运算符。特别是赋值运算符 (operator=) 更容易出错，更应该避免重载。如果需要你可以定义 Equals() 和 CopyFrom() 函数。同时，如果一个类有一丁点可能被前置声明，那么在任何情况下都要避免重载危险的一元操作符 operator&。

不要重载 operator""，即不要引入用户自定义字面量。

然而，还是有极少的情况你需要重载一个运算符以便与模板或 ``标准"C++ 类（如用以打印日志的 operator<<(ostream&, const T&)）交互。只有有充分正当的理由才能重载，但你还是要尽量避免这种情况。特别不要为了使你的类可以作为 STL 容器的键值而重载 operator== 和 operator<，这时，你应该在声明容器时创建判断相等和比较大小的仿函数类型。

一些 STL 算法要求你重载 operator==，这时你可以这样做，但是要在文档中说明原因。

参考拷贝构造函数和函数重载。

4.11 存取控制

数据成员都定义成私有的，如果需要就提供存取函数来访问（因为技术原因，当使用 **Google Test** 时，允许夹具类的数据成员是 **pretected** 的）。通常一个名为 **foo_** 的成员，存取函数名为 **foo()**，可能还要一个设值函数 **set_foo()**。例外：静态常量数据成员不需要 (通常称为 **kFoo**) 是私有的。

存取函数在头文件中内联定义。

参见继承和函数命名。

4.12 声明顺序

在类中使用特定的声明顺序：**public:** 在 **private:** 前，方法在数据成员（变量）前，等。

你的类应该以 public: 区开始，然后是 protected: 区，然后是 private: 区。如果某区是空的，就忽略它。

每一区中，声明通常应该按如下的顺序：

- 类型定义（typedef）和枚举
- 常量（静态常量成员）
- 构造函数
- 析构函数
- 方法，包括静态方法
- 数据成员（除子静态常量成员）

友元声明应该总是在 private 区，DISALLOW_COPY_AND_ASSIGN 宏应该在 private: 区最后，它应该是一个类的末尾。参见拷贝构造函数。

相应的.cc 文件中方法定义顺序应该尽量和头文件中的保持一致。

大的方法定义不要内联在类定义中。通常，那些没有特别意义或对性能要求高，还非常短小的方法才可以内联。参见内联函数。

4.13 写短小函数

尽量编写短小精炼的函数。

我们意识到有时候长函数更合适，所以在函数长短方面没有硬性的规定。如果一个函数超过 40 行，就考虑一下在不破坏程序结构的情况下能否将其拆分。

即使你的长函数现在可以完美工作，几个月后可能会有人为其添加新的行为，这可能引入难以查找的 bug。保持函数短小、简单可以使他人更容易阅读和修改你的代码。

你也许会在一些代码中发现一些又长又复杂的函数。不要被修改已有代码吓倒：如果和复杂的函数打交道，你发现错误难以调试，或是你只想使用一部分代码，你可以考虑把这个函数拆成可控的小函数。

5 Google 的奇技淫巧

我们使用了大量技巧和工具使 C++ 代码更健壮，我们使用 C++ 的方式可能与你在别处见到的有所不同。

5.1 所有权和智能指针

对动态分配的对象优先使用单独的、固定的拥有者。优先使用智能指针转移所有权。

5.1.1 定义：

``所有权" 是一种用以管理动态分配的内存（还有其它资源）的记账技术。动态对象的拥有者是一个对象或函数，它们要负责在对象不再需要时删除它们。所有权有时可以共享，这时通常是最后一个拥有者负责删除。即使所有权不共享，它也可以从一段代码转移到另一段。

智能指针是像指针一样工作的类，如，通过重载 * 和 ->。一些智能指针类型可用以自动化所有权的记账，以确保完成其职责。std::unique_ptr 是一个 C++11 引入的智能指针，表示动态对象的独占所有权；当 std::unique_ptr 离开其作用域时，对象被删除。它不能被拷贝，但可以 move 以体现所有权转移。shared_ptr 是一个表示动态对象共享所有权的智能指针。shared_ptr 可以拷贝，所有的拷贝共享所有权，当最后一个 shared_ptr 销毁时对象被删除。

5.1.2 优点：

- 不使用一些所有权逻辑，管理动态分配的内存几乎不可能。
- 转移一个对象的所有权比拷贝它要便宜（如果可以拷贝的话）。
- 转移一个对象的所有权比 ``借" 一个指针或引用要简单，因为它减少了两个用户之间关于此对象生命周期的交互。
- 智能指针通过显式的所有权逻辑、自文档且无歧义提高了可读性。
- 智能指针可以消除手动所有权记录，简化代码从而排除一大类错误。
- 对于常量对象，共享所有权比深度拷贝更简单也更高效。

5.1.3 缺点：

- 所有权必须通过指针（不管是智能指针还是普通指针）来表达和转移。指针的语义比值要复杂得多，特别是在 API 中：你要关心的不只是所有权，还有别名、生命周期、可变性以及其它事项。
- 值语义的性能开销经常被高估，所以所有权转移的性能优势可能不足以弥补可读性和复杂性的损失。
- 所有权转移 API 把用户绑定在了单一内存管理模型上。
- 使用智能指针的代码在资源是否已释放这事上更不明确。

- std::unique_ptr 使用了 C++11 的 move 语义来表达所有权转移，在 Google 代码中这通常是被禁止的，可能会迷惑一些程序员。
- 共享所有权比小心地所有权设计更有诱惑力，会使系统设计更模糊。
- 共享所有权需要在运行时显式记账，可能会有开销。
- 有些情况下（如，循环引用），有共享引用的对象可能永不会删除。
- 智能指针不是纯指针的完美替代。

5.1.4 结论：

如果动态分配是必须的，最好把所有权控制在执行分配的代码中。如果其它代码需要访问这个对象，考虑传递一个拷贝、一个指针或引用给它，而不是使用所有权转移。优先使用 std::unique_ptr 来进行所有权转移。如：

```
std::unique_ptr<Foo> FooFactory();
void FooConsumer(std::unique_ptr<Foo> ptr);
```

没有很好的理由不要把代码设计成使用共享所有权。其中一个理由是避免昂贵的拷贝操作，但只有在性能很关键且相关对象是不可变的（即，shared_ptr<const Foo>）时你才应该使用。如果非要使用共享所有权，优先使用 shared_ptr。

不要在新代码中使用 scoped_ptr，除非你需要兼容老版本的 C++。永远不要使用 linked_ptr 和 std::auto_ptr。这三种情况下，都应该使用 std::unique_ptr。

5.2 cpplint

使用 **cpplint.py** 检查风格错误。

cpplink.py 是一个读取源代码并能指出许多风格错误的工具。它并不完美，有时会漏报或误报，但是依然是个有价值的工具。误报可以行尾加// NOLINT 注释来忽略。

一些项目会指导你使用该项目的工具运行 cpplint.py。如果你的项目没有，可以单独下载。

6 其它 C++ 特性

6.1 引用参数

所有以引用方式传递的参数都要是 **const** 的。

6.1.1 定义：

在 C 语言中，如果一个函数要修改一个变量，这个参数就必须是一个指针，如 `int foo(int *pval)`。在 C++ 中，有了新的选择就是可以声明一个引用参数：`int foo(int &val)`。

6.1.2 优点：

把参数定义成引用参数可以避免像 `(*pval)++` 这样丑陋的代码。对于像拷贝构造函数这样应用也是必须的。有一点要说清楚，引用不像指针，不能有空引用。

6.1.3 缺点：

引用容易引起混乱，因为它有值的语法但语义却是指针。

6.1.4 结论：

函数的参数列表中，所有的引用参数都必须是 `const` 的：

```
void Foo(const string &in, string *out);
```

事实上在 Google 的代码这是一个硬性规定：输入参数是值或 `const` 引用，而输出参数是指针。输入参数可以是 `const` 指针，但是非 `const` 的引用是决不允许的，除非有约定明确要求，比如 `swap()`。

然而，确实有时候用 `const T*` 比 `const T&` 更合理。如：

- 你想传递一个空指针
- 函数把一个指针或引用保存到输入

记住，大多数情况下输入参数都必须声明成类似 `const T&` 这样。使用 `const T*` 是告诉看代码的人这个输入参数被特别对待了。所以一旦你选择使用 `const T*` 而非 `const T&`，一定要有具体的理由，否则就会误导读者去查找一个不存在的解释。

6.2 右值引用

不要使用右值引用、**`std::forward`**、**`std::move_iterator`** 或 **`std::move_if_noexcept`**。只在参数不可拷贝时使用 **`std::move`** 的单参数形式。

6.2.1 定义：

右值引用是一种只能绑定到临时对象的引用。语法和传统引用类似。如，`void f(string&& s);` 声明了一个函数，其参数为一个到字符串的右值引用。

6.2.2 优点：

- 定义一个 `move` 构造函数（一个接收类的右值引用的构造函数）使得可以使用 `move` 代替拷贝。例如，如果 `v1` 是一个 `vector<string>`，那么 `auto v2(std::move(v1))` 多半仅仅就是简单的指针维护而不是拷贝大量数据。有时这可以大幅提高性能。

- 使用右值引用才可以写出将其参数传递给另一个函数的泛型函数，且无论参数是否是临时对象都可以工作。

- 使用右值引用才能实现可移动但不能拷贝的类型，这对一些类型很有用，这些类型没有合理的拷贝定义，但你可能想把它们作为参数传递给一个函数，或把它们放在容器中，等等。

- 要高效使用某些标准库类型，如 `std::unique_ptr`，`std::move` 是必须的。

6.2.3 缺点：

- 右值引用是一个相对较新的特性（作用 C++11 的一部分引入），还没有被广泛理解。像引用折叠和 `move` 构造函数的自动合成等规则都很复杂。

- 右值引用鼓励一种重度使用值语义的编程风格。这种风格许多开发者都不熟悉，且其性能特性很难推断。

6.2.4 结论：

不要使用右值引用，也不要使用辅助函数 `std::forward` 或 `std::move_if_noexcept`（它们本质上只是转换成右值引用类型），或 `std::move_iterator`。只在参数不可拷贝（如 `std::unique_ptr`）时，或在对象可能不会被拷贝的模板代码中，才使用 `std::move` 的单参数形式。

6.3 函数重载

使用重载函数（包括构造函数）时，一定要确保看代码的人一看到调用就能知道发生了什么，而不用先去查找哪一个重载函数被调用了。

6.3.1 定义：

你可以写一个以 `const string&` 为参数的函数，然后再写一个以 `const char *` 为指针的。

```
class MyClass {
 public:
  void Analyze(const string &text);
  void Analyze(const char *text, size_t textlen);
};
```

6.3.2 优点：

通过参数不同的同名函数，重载可以使代码更直观。模板化代码要求重载，这也方便了代码使用者。

6.3.3 缺点：

如果一个函数只是以参数不同重载，读者可能需要了解 C++ 复杂的匹配规则来确定将要发生什么。同样，如果继承时一个派生类只重载了函数的一些变体，会使许多人找不着北的。

6.3.4 结论：

当你要重载一个函数时，考虑一下让函数含有一些参数信息是不是更好，如，用 `AppendString()`，`AppendInt()` 代替重载 `Append()`。

6.4 缺省参数

除了下面介绍的有限的几种情况，我们不允许缺省参数的函数。如果合适，就用函数重载来模拟。

6.4.1 优点：

你的函数经常会用到默认值，但你偶尔会想要重载这些默认值。使用缺省参数可以轻松搞定，又不必为了这种少数例外定义多个函数。相对于重载函数，缺省参数语法更干净，样板代码更少，可以更清楚地区分 `必须的' 和 `可选的' 参数。

6.4.2 缺点：

有缺省参数的函数指针容易引起混乱，因为函数签名经常和调用签名不匹配。给一个已存在的函数添加一个缺省参数会改变它的类型，对使用其地址的代码来说这可能会有问题。添加一个函数重载就没有这些问题。另外，缺省参数可能使代码变笨重，因为它们在每个调用处都复制一份 --与重载函数不同，``默认值" 只出现在函数定义处。

6.4.3 结论：

尽管上面的缺点并不繁重，但是仍然比相对函数重载的那点好处要大。所以除了下面的情况外，我们要求所有的参数都必须明确指定。

一个特例是当这个函数是一个.cc 文件中的静态函数（或在一个匿名命名空间中）时。这时没有上面的缺点，因为函数的使用被本地化了。

另一个特例是缺省参数用以模拟可变参数列表。

```
// 使用缺省的空 AlphaNum 参数，最多可以支持 4 个参数
string StrCat(const AlphaNum &a,
              const AlphaNum &b = gEmptyAlphaNum,
```

```
    const AlphaNum &c = gEmptyAlphaNum,
    const AlphaNum &d = gEmptyAlphaNum);
```

6.5 变长数组和 alloca()

我们不允许使用可变长度数组和 **alloca()**。

6.5.1 优点：

可变长度数组有自然的语法。可变长度数组和 alloca() 都非常高效。

6.5.2 缺点：

可变长度数组和 alloca 都不属于标准 C++。更重要的是，它们根据数据大小分配栈空间，这可能引发难以发觉的内存越界复写 bug：``代码在我的机器上没问题，可是在产品中却会莫名其妙地挂掉"。

6.5.3 结论：

要使用安全的内存分配器，如 scoped_ptr/scope_array。

6.6 友元

我们允许合理使用友元类和函数。

友元通常应该定义在同一个文件中，这样读者就不用为了查看这个私有成员的用法而查找另一个文件。友元的一个常用用法是：将 FooBuilder 类定义成 Foo 类的友元来正确构造 Foo 类的内部状态，又不用将内部状态暴露给外面。有时将单元测试类定义成被测试的类的友元很有用。

友元扩展而非破坏了类封装的边界。在一些情况下，如果你只是想在另一个类中访问一个类，友元比使用公有成员更好。当然，大多数类应该通过公有成员来与其它类交互。

6.7 异常

我们不使用异常。

6.7.1 优点：

- 异常允许在应用程序的更上层决定如何处理那些深度嵌套的函数中 ``不可能发生的" 错误，不会像记账式的错误处理代码那样含糊又容易出错。

- 许多现代语言都使用异常。在 C++ 中使用异常会和 Python，Java 以及其它人熟悉的 C++ 保持一致。

- 一些第三方 C++ 库使用异常，在内部关闭异常将难以与这些库集成。

- 异常是构造函数报告失败的唯一方法。我们可以使用工厂函数或 `Init()` 方法来模拟，但是它们又分别需要堆内存分配或一个新的 ``无效" 状态。

- 异常在测试框架中相当方便。

6.7.2 缺点：

- 当给一个已存在的函数添加一个 `throw` 语句时，你必须测试调用路径上的所有调用者。它们或者能至少保证基本的异常安全，或者不捕获异常并乐于接受程序退出的结果。举个例子，如果 `f()` 调用了 `g()`，后者又调用了 `h()`，`f` 捕获了 `h` 抛出一个异常，这时 `g` 要特别小心，否则就容易清理不妥当。

- 更普遍的情况是异常使靠看代码弄明白程序的控制流非常困难：函数可能会在你想不到的地方返回。这引发维护和调试困难。你可能通过一些如何使用异常的规则来最小化这种代价，但代价是使开发者要理解更多的东西。

- 异常安全需要 RAII 和不同的编码实践。需要大量的支持机制才能轻易写出异常安全的代码。为了避免要求阅读者理解整个调用关系图，异常安全的代码必须把写入持久状态的逻辑隔离到 ``提交" 阶段。这有好处也有代价（可能为了隔离提交你不得不弄乱代码）。允许异常使我们无论是否值得都要付出这些代价。

- 使用异常会向生成的二进制文件中添加数据，增加了编译时间（或许微不足道），并增加了地址空间的压力。

- 可以使用异常会使开发者在不恰当的时候抛出，又在不安全的时候恢复某些异常。如，无效的用户输入不应该引发异常。要写下这些限制，这份指南会长很多。

6.7.3 结论：

表面上使用异常利大于弊，特别是对新项目。但对于已有代码，引入异常会影响所有依赖的代码。如果异常可以向新项目以外传播，在跟之前未使用异常的代码集成时也会碰到麻烦。因为 Google 中大部分 C++ 代码都没有准备处理异常，引入新的产生异常的代码更是困难。

由于 Google 的已有代码不使用异常，使用异常的代价比新项目要高。迁移的过程会很慢并容易出错。我们不相信异常的有效替代，如错误处理代码和断言，会引入严重的负担。

我们反对使用异常的建议不是出于哲学或道德判断，纯粹是出自于实践。因为我们希望在 Google 使用我们的开源项目，但项目中使用异常会很麻烦，因此我们也建议在 Google 的开源项目中不使用异常。如果一切都从头开始，可能会有所不同。

这条禁令也适用于 C++11 中加入的异常相关的特性，如 `noexcept`，`std::exception_ptr`，和 `std::nested_exception`。

对 Windows 代码这条规则有一个例外（没有双关的意思）。

> 我说：看来这条规则主要是出于和 Google 的遗留代码兼容。实际中是不是要用再判断吧。

6.8 运行时类型信息（RTTI）

避免使用运行时类型信息（**RTTI**）。

6.8.1 定义：

RTTI 允许程序员在运行时查询一个对象的类型。这是由 `typeid` 和 `dynamic_cast` 实现的。

6.8.2 缺点：

运行时查询一个对象的类型通常意味着设计有问题。需要在运行时知道一个对象的类型往往说明你的类继承层次设计有缺陷。

无原则地使用 RTTI 会使代码难以调试。会导致代码中充斥着基于类型的决策树或 `switch` 语句，以后要修改时，这些都是必须要检查的。

6.8.3 优点：

RTTI 的标准替代品（下面会描述）需要修改或重新设计类的继承结构。有时修改是不可行或不能接受的，特别是那些已经被广泛使用或很成熟的代码。

RTTI 在一些单元测试中会很有用。例如，测试工厂类时需要校验新生成的对象是否是预期的类型，这时 RTTI 就很有用。在管理对象和其 mock 对象的关系时也很有用。

当有多个抽象对象时，RTTI 很有用。考虑下面代码：

```
bool Base::Equal(Base* other) = 0;
bool Derived::Equal(Base* other) {
  Derived* that = dynamic_cast<Derived*>(other);
  if (that == NULL)
    return false;
  ...
}
```

6.8.4 结论：

RTTI 有合理的应用，但是容易滥用，所以使用时你必须要小心。在单元测试时你可以放心使用 RTTI，但是其它代码中应尽可能避免。当你发现你写的代码需要根据对象类型采取不同的动作时，考虑下面的方案来代 RTTI：

- 虚方法是根据不同的子类执行不同代码的优选方案。这样就把工作放在对象内部了。
- 如果工作属于对象外的某些处理代码，考虑使用如同观察者模式这样的二次分发解决方案。它允许对象外的设施可以使用内建类型系统来确定对象类型。

如果程序逻辑确保了一个给定的基类实例实际上是一个特定子类的实例，这时可以使用 `dynamic_cast`。通常这种情况可以使用 `static_cast` 代替。

基于类型的决策树是你代码有问题的强有力的信号。

```
if (typeid(*data) == typeid(D1)) {
  ...
} else if (typeid(*data) == typeid(D2)) {
  ...
} else if (typeid(*data) == typeid(D3)) {
...
```

上面这样的代码通常会在类的继承层次上添加了额外的子类后出错。此外，当一个子类的属性发生变化时，很难修改所有受影响的代码段。

不要去实现一个类似 RTTI 的解决方案。我们反对 RTTI 的理由同样适用于在类的继承层次中使用类型标签的方案。此外，这样的解决方案会掩盖你真实的意图。

6.9 强制类型转换

要使用 C++ 式的强制类型转换，如 **`static_cast<>()`**。不要使用其它的类型转换形式，如 **`int y = (int)x;`** 或 **`int y = int(x);`**。

6.9.1 定义：

C++ 引入了与 C 不同的类型转换系统，会区分转换操作的目标类型。

6.9.2 优点：

C 语言类型转换的问题是模棱两可，有时是值转换（conversion, 如，`(int)3.5`），有时是强制类型转换（cast，如，`(int)"hello"`），而 C++ 的类型转换避免了这种情况。另外 C++ 的类型转换很容易查找。

6.9.3 缺点：

语法挺恶心的。

6.9.4 结论：

要使用 C++ 风格的类型转换，而不是使用 C 风格的。

- `static_cast` 可以作为 C 风格值转换的替代，或用来将指针提升为其父类的指针。
- 使用 `const_cast` 来去掉 `const` 修饰符（见 const）。

- 使用 `reinterpret_cast` 来执指针类型和整数或其它类型指针之间接非安全转换。使用时一定要清楚你在干什么，并对明白 aliasing 问题（见下）。

对 `dynamic_cast` 的使用指南参见 RTTI 一节。

> aliasing 在 C/C++ 中指 ``不同类型的指针指向同一地址"。

6.10 流

只在记日志时才使用流。

6.10.1 定义：

流是 `printf()` 和 `scanf()` 的替代。

6.10.2 优点：

使用流，你就不必知道要打印的对象类型。不会有格式化字符串和参数列表不匹配的问题。（使用 gcc 时，`printf` 也没有这个问题。）流的构造函数和析构函数会自动打开和关闭相关文件。

6.10.3 缺点：

流难以实现类似 `pread()` 的功能。不用 printf 类似的手段，用流难以高效实现一些格式化操作（特别是常用的格式化字符串`%.*s`）。流不支持重排操作（`%1s` 指令），而这对于国际化很有用。

6.10.4 结论：

除非在日志接口中，否则不要使用流，而应该使用 `printf` 风格的函数。

使用流有好处也有坏处，但是又一次，一致性胜过一切。不要在代码中使用流。

6.10.5 扩展讨论：

关于这个问题有一些争论，所以在此进一步解释一下。回想一下 唯一性（**Only One Way**）原则：我们希望确保使用同一类型的 I/O 的代码看起来都是一样的。因此，我们不想让用户来决定使用流还是 `printf+read/write` 等。相反，我们应该明确唯一一种方式。之所以日志例外，因为它是很特别的应用，并有一些历史原因。

流的支持者们认为流是不二之选，但是理由实际上都不怎么清楚。流的每一个优点都有相应的劣势。最大的优势莫过于你无需关心要打印的对象类型，这的确是优势。但是也有不利的一面：你容易用错类型，并且编译器不会警告你。使用流，你很容易犯下面的错误而不自知：

```
cout << this;    // 打印地址
cout << *this;   // 打印内容
```

因为 << 已被重载，所以编译器不会报错。就是因为这原因，我们才不鼓励重载。

有人说 printf 风格丑陋难以阅读，便流也好不到哪里去。考虑下面两段代码，以两种风格实现相同功能。哪一个更清晰？

```
cerr << "Error connecting to '" << foo->bar()->hostname.first
     << ":" << foo->bar()->hostname.second << ": " << strerror(errno);

fprintf(stderr, "Error connecting to '%s:%u: %s",
        foo->bar()->hostname.first, foo->bar()->hostname.second,
        strerror(errno));
```

这样的例子可以举出很多。（你可能会争论说 ``合理封装一下会更好"，但是这里可以，其它地方呢？还有，记住，我们的目标是使语言更小，而不是添加更多需要人去学习的新设施。）

两种方式都有各自的优点和缺点，并且没有一个超级解决方案。简单原则让我们必须选择其一，多数的决定是 printf+read/write。

6.11 前置自增和自减

迭代器和其它模板对象的自增和自减要使用前缀形式（**++i**）。

6.11.1 定义：

当一个对一个变量进行自增 (++i 或 i++) 或自减 (--i 或 i--) 后，表达式的值又没有被用到，就必须确定是用前置操作还是后置操作。

6.11.2 优点：

当返回值被忽略时，前置形式（++i）至少不会比后置形式（i++）效率低，通常会高得多。因为后置操作需要一个 i 的拷贝作为表达式的返回值。当 i 是迭代器或其它非数值类型时，拷贝 i 可能开销很大。既然当返回值被忽略时这两种形式作用一样，那为什么不总是使用前置操作？

6.11.3 缺点：

传统的 C 语言开发中，当表达式值没有被用到时通常使用后置操作，特别是在 for 循环中。有人觉得后置操作更易读，因为 ``主语"(i) 执行了 ``动作"(++) 这种结构更像英语。

6.11.4 结论：

对于简单的值类型（不是对象），用什么都无所谓。对于迭代器和其它模板类型，必须使用前置操作。

6.12 const 的使用

合理使用 **const**。**C++11** 中，对有些常量的使用 **constexpr** 是更好的选择。

6.12.1 定义：

声明变量和参数时可以使用 const 修饰以表明此变量不会被修改（如 const int foo）。对类方法使用 const 标识符表明此方法不会修改类的成员变量（如，class Foo { int Bar(char c) const; };）。

6.12.2 优点：

使人很容易就知道变量是如何使用的。使编译器可以更好地做类型检查，自然也会生成更好的代码。帮助写出正确的代码，因为人们知道他们调用的函数在是否能修改他们的变量方面是如何受限的。还可以帮助人们知道在多线程程序中哪些函数是可以无锁安全调用的。

6.12.3 缺点：

const 是有传染性的：如果你传递一个 const 变量给一个函数，这个函数的原型必须也要接收 const 变量（否则这个变量就需要去 const 强制类型转换）。这个问题在调用库函数时尤为麻烦。

6.12.4 结论：

const 变量，数据成员，方法和参数添加了一层编译期类型检查，可以尽快查错。因此我们强烈建议你在所有合适的情况下使用 const：

- 如果函数不会修改一个引用或指针参数指向的值，这个参数应该是 const 的。
- 尽可能把方法声明成 const 的。访问函数总应该是 const 的。如果不修改类的数据成员，没有调用非 const 方法，并且没有返回非 const 值或数据成员的非 const 引用，方法就应该是 const 的。
- 构造完对象之后就不会再修改的数据成员也考虑声明成 const 的。

允许使用 mutable 关键字，但多线程时就不安全了，所以应该首先认真考虑线程安全问题。

6.12.5 何处放置 const

相对于 const int* foo, 有人喜欢更 int const *foo。认为后者更可读，因为更一致：它遵循了 const 总是紧跟着它修饰的对象的原则。但是这种一致性在没有深度嵌套的指针表达式时并不适用，因为多数的 const 表达式中都只有一个 const，作用于基本值。这时，就没有所谓的一致性要维护了。也可以说将 const 放在最前面更易读，因为更符合英语习惯：将 ``形容词"(const) 放在 ``名词"（int）前。

简而言之，我们鼓励将 const 放在最前面，但不强制要求。只要你自己的代码保持一致就行。

6.13 constexpr 的使用

C++11 中，使用 constexpr 来定义真正的常量或保证常量初始化。

6.13.1 定义：

一些变量可以声明成常量以表明这些变量是真正的常量，即，在编译期和链接期都是固定的。

6.13.2 优点：

`constexpr` 的使用使得可以使用浮点数表达式定义常量而不仅仅是字面量；可以定义用户自定义类型的常量；还有和函数调用一起的常量定义。

6.13.3 缺点：

过早地把一些东西使用 `constexpr` 标记在稍后想要降级的时候可能会引起迁移问题。目前 `constexpr` 函数和构造函数中的限制可能会在这些定义中引入晦涩的解决方案。

6.13.4 结论：

`constexpr` 定义使接口的常量部分有了一个更稳健的规范。使用 `constexpr` 来指定真正的常量和支持它们定义的函数。为了支持 `constexpr`，要避免复杂的函数定义。不要使用 `constexpr` 来强制内联。

6.14 整数类型

在 **C++** 内建的整数类型中，只使用 **`int`**。如果程序需要不同大小的变量，就使用 **`<stdint.h>`** 中有明确宽度的整数类型，如 **`int16_t`**。如果你的变量表示的值可能大于或等于 **2^31**（**2GiB**），就使用 **64** 位类型如 **`int64_t`**。记住即使你的值不会比 **`int`** 大，它还可能用于计算的中间结果，而中间结果有可能需要更大的类型。当不确定时，就选择一个更大的类型。

6.14.1 定义：

C++ 没有指定其整数类型的大小。通常都假设 `short` 是 16 位，`int` 是 32 位，`long` 是 32 位，`long long` 是 64 位的。

6.14.2 优点：

声明一致。

6.14.3 缺点：

C++ 中整数类型的大小可能随编译器和体系结构不同而有所不同。

6.14.4 结论：

<stdint.h> 中定义了 int16_t，int32_t，int64_t 等类型。当你需要保证一个整数的大小时，你应该使用这些类型，而非 short，unsigned long long 类似的类型。C 语言中的整数类型，只能使用 int。在合适的情况下，推荐使用像 size_t 和 ptrdiff_t 这样的标准类型。

对于已知不会太大的整数，我们最经常使用的就是 int，如循环计数器等。类似的情况就用原生的 int 。你可以假设 int 最少是 32 位的，但不要假设它会比 32 位更大。如果你需要 64 位整数，应该用 int64_t 或 uint64_t。

对于那些可能会 ``很大" 的整数，用 int64_t。

除非你有合理的理由，如要表示一个位图而非一个数，或需要定义溢出模 2^N , 否则不要使用无符号数。尤其不要因为一个数不可能为负而使用无符号类型，这时应该使用断言。

如果你的代码是一个可以返回大小的容器，要确保使用可以适应所有可能情况的类型。不确定时，就使用选大的那个类型。

整数类型互相转换时要小心。整数转换和提升可以会引发难以捉摸的行为。

6.14.5 关于无符号整数

有些人，包括一些教科书作者，都推荐使用无符号类型来表示永远不可能为负的数，以图达到代码的自文档化。然而在 C 语言中，这种好处被可能引入的真实 bug 掩盖。考虑下面的代码：

```
for (unsigned int i = foo.Length()-1; i >= 0; --i) ...
```

这段代码永远都不会结束！有时 gcc 会发现并提醒你，便是通常都不会。同样的 bug，在比较有符号和无符号数时也会产生。基本上，是 C 语言的类型提升机制使无符号类型的行为不同于预期。

所有使用断言来文档化一个变量不能为负，不要使用无符号类型。

6.15 64 位移植性

代码应该是 64 位和 32 位都支持的。要时刻考虑到打印，比较和结构体对齐等问题。

- printf() 的有些指示符不能在 32 位和 64 位之间很好的移植。C99 定义了一些可移植的指示符。不幸的是 MSVC7.1 支持得不全，且标准本身也有所遗漏，所以我们有时不得不定义自己的丑陋版本（按标准头文件 inttypes.h 的风格）。

```
// size_t 的 printf 的宏，按 inttypes.h 风格
#ifdef _LP64
#define __PRIS_PREFIX "z"
#else
#define __PRIS_PREFIX
#endif

// 在 printf 中格式化字符串的% 后使用这些宏，在 32/64 位下都可以获得正确行为，像这样：
```

```
// size_t size = records.size();
// printf("%"PRIuS"\n", size);

#define PRIdS __PRIS_PREFIX "d"
#define PRIxS __PRIS_PREFIX "x"
#define PRIuS __PRIS_PREFIX "u"
#define PRIXS __PRIS_PREFIX "X"
#define PRIoS __PRIS_PREFIX "o"
```

类型	不要使用	使用	备注
void *(或任何指针类型)	%1x	%p	
int64_t	%qd,%lld	%``PRId64"	
uint64_t	%qu,%llu,%llx	%``PRIu64", %``PRIx64"	
size_t	%u	%``PRIuS", %``PRIxS"	C99 中是%zu
ptrdiff_t	%d	%``PRIdS"	C99 中是%td

注意 `PRI*` 这些宏展成后会由编译器拼接成独立字符串。因此如果你使用非常量的格式化字符串，应该插入值而非宏名。仍然可以包含长度指示符，如使用 `PRI*` 宏时在`%` 后面。综上所述，举个例子，`printf("x=%30"PRIuS"\n",x)` 在 32 位 Linux 上会展开成 `printf(x=%30" "u" "\n", x)`，编译器看来就是 `printf("x = %30u\n",x)`。

- 记住 `sizeof(void *)` 不等于 `sizeof(int)`。如果需要和指针一样大小的整数，需要使用 `intptr_t`。

- 你要小心结构体对齐，特别是对那些需要保存在磁盘上的结构体。在 64 位系统上，任何含有 `int64_t` 或 `uint64_t` 数据成员的类或结构体都会以 8 字节对齐。如果你需要 32 位代码和 64 位代码在磁盘上共享它，就必须要确保在两种架构上以一致的方式打包。大部分编译器都提供了改变结构体对齐方式的方法。gcc 可以使用 `__attribute__((packed))`，MSVC 则提供了 `#pragma pack()` 和 `__declspec(align())`。

- 创建 64 位常量时要使用 `LL` 或 `ULL` 后缀。如：

```
int64_t my_value = 0x123456789LL;
uint64_t my_mask = 3ULL << 48;
```

- 如果确实需要在 32 位和 64 位使用不同的代码，使用 `#ifdef _LP64` 来区分。（但是应尽量避免这样使用，应保持代码修改的局部化。）

6.16 预处理宏

使用宏要特别小心。优选函数、枚举和 **`const`** 变量。

宏意味着你看到的代码和编译器看到的代码是不一样的。这可能引入非预期的行为，特别是当宏拥有全局作用域时。

庆幸的是，宏在 C++ 中不像在 C 中那样必须。当宏用以内联性能关键的代码时，可以使用内联函数；当宏用以保存常量时，可以使用 `const` 变量；当宏用以``缩写"一长变量名时，可以使用引用；当使用宏来条件编译代码时。。。好吧，避免那样做（当然有例外，就是头文件使用 `#define` 保护以避免重复包含）。宏让测试变得异常困难。

宏可以做一些其它技术做不到的事，你可以在一些代码库，尤其中底层库中看到。其中有些特性（像字符串化（`#`），和连接 `##` 等）都不能通过语言本身实现。但决定使用宏之前，一定要认真考虑是否有其它替代方案。

以下使用模式可以避免宏的许多问题，如果你使用宏，请尽量遵循以下几条：

- 不要在.h 文件中定义宏。
- 紧挨着使用之前定义宏，用完马上 `#undef` 掉。
- 在替换成你自己的宏之前不要只是简单 `#undef` 掉已有的宏，你应该选择一个看起来不会冲突的名字。
- 不要使用展开后会破坏 C++ 代码结构的宏，至少也应将行为用文档详细描述。
- 不要使用 `##` 来生成函数/类/变量名.

6.17 0 和 nullptr/NULL

整数用 **`0`**，实数用 **`0.0`**，指针用 **`nullptr`** 或 **`NULL`**，字符（串）用**`'\0'`**。

整数用 `0`，实数用 `0.0`。这两点没有争议。

对于指针（地址值），可以在 `0`、`NULL` 和 `nullptr` 中选择。对于允许使用 C++11 标准的工程，使用 `nullptr`。对于 C++03 的工程，我们使用 `NULL`，因为它看起来更像个指针。实际上，一些编译器提供了特殊定义的 `NULL`，以给出有用的警告信息，尤其中 `sizeof(NULL)` 和 `sizeof(0)` 不相等的情况。

对字符（串）使用 `\0`。这是正确的类型同时使代码更易读。

6.18 sizeof

相对于 **`sizeof(`**类型**`)`**，优选 **`sizeof(`**变量名**`)`**。

当你想要一个特定变量的大小时，使用 `sizeof(`变量名`)`。`sizeof(`变量名`)` 会随变量类型的变化更新。在与特定变量无关时你才可以使用 `sizeof(`类型`)`，如在管理外部或内部数据格式的代码中，就不方便使用相应 C++ 类型的变量。

```
Struct data;
memset(&data, 0, sizeof(data));    // 好!
```

```
memset(&data, 0, sizeof(Struct)); // 不好!

if (raw_size < sizeof(int)) { // 可以
  LOG(ERROR) << "compressed record not big enough for count: " << raw_size;
  return false;
}
```

6.19 auto

只对于那些凌乱的类型名才使用 **auto** 避免。只要有利于可读性就继续使用显式的类型声明，永远不要在局部变量之外使用 **auto**。

6.19.1 定义：

在 C++11 中，一个 auto 类型的变量的实际类型会和初始化它的表达式匹配。你可以在以拷贝方式初始化变量时，或绑定引用时使用 auto。

```
vector<string> v;
...
auto s1 = v[0];          // 生成一个 v[0] 的拷贝
const auto& s2 = v[0];   // s2 是 v[0] 的引用
```

6.19.2 优点：

C++ 的类型名有时非常长且笨重，特别是涉及模板和命名空间时更是如此。在下面这样的语句中：

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

返回类型难以阅读，并掩盖了代码的主要目的。改成下面这样就好读多了：

```
auto iter = m.find(val);
```

没有 auto，在有些表达式中我们要写两次类型名，而这对于代码阅读者没有任何价值，如

```
diagnostics::ErrorStatus* status = new    diagnostics::ErrorStatus("xyz");
```

auto 使得合理使用中间变量变得更容易，减少了显示写出它们类型的麻烦。

6.19.3 缺点：

有时使用显式的类型使代码更清晰，特别是当一个变量初始化时要依赖远处声明的东西时。像在下面的表达式中：

```
auto i = x.Lookup(key);
```

如果 x 是在几百行代码之前声明的，i 的类型就不明显了。

程序员需要理解 auto 和 const auto& 的不同，否则就会在不必要的地方发生拷贝。

auto 和 C++11 中的大括号初始化一起用可能会引起困惑。下面的声明是不同的：

```
auto x(3);  // Note: parentheses.
auto y{3};  // Note: curly braces.
```

x 是一个 int，而 y 是一个 initializer_list。通常不可见的代理类型也有同样的问题。

如果 auto 变量是接口的一部分，如，是头文件中的常量，程序员就可能在改变其值时无意中改变了其类型，这会导致非预期的 API 剧烈变化。

6.19.4 结论：

auto 只对局部变量是允许的。对文件作用域或命名空间作用域的变量，以及类的成员变量都不要使用 auto。不要把大括号初始化列表赋值给一个 auto 变量。

auto 关键字还被用在一个无关的 C++ 特性中：它是以 ``trailing return type"（延时判断函数返回值类型）方式声明函数语法的一部分。以 ``trailing return type" 方式声明函数是不允许的。

6.20 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（没有构造函数的数组和结构体）可以使用大括号初始化。

```
struct Point { int x; int y; };
Point p = {1, 2};
```

C++11 把这种语法扩展到了所有数据类型，这种形式被称为 brace-init-list。下面是一些使用示例。

```
// 包含几个元素的 Vector
vector<string> v{"foo", "bar"};

// 和上面一样，这种形式要求 initializer_list 的构造函数不能是 explicit 的。
// 否则你应该选择其它形式。
vector<string> v = {"foo", "bar"};
```

```
// 包含 pair 列表的 Map。嵌套的 braced-init-lists 也能工作。
map<int, st5rring> m = {{1, "one"}, {2, "2"}};

// braced-init-list 可以隐式转换成返回值类型。
vector<int> test_function() {
  return {1, 2, 3};
}

// 遍历 braced-init-list.
for (int i : {-1, -2, -3}) {}

// 用 braced-init-list 调用函数
void test_function2(vector<int> v) {}
test_function2({1, 2, 3});
```

自定义数据类型也可以定义使用 `initializer_list` 的构造函数，它会自动从 braced-init-list 创建：

```
class MyType {
 public:
  // initializer_list 是底层初始化列表的引用，所以可以传值
  MyType(initializer_list<int> init_list) {
    for (int element : init_list) {}
  }
};
MyType m{2, 3, 5, 7};
```

最后，大括号初始化也可以在没有 `initializer_list` 构造函数时调用普通构造函数。

```
double d{1.23};
// 只要 MyOtherTypeinitializer_list 没有构造函数，就调用普通构造函数。
class MyOtherType {
 public:
  explicit MyOtherType(string);
  MyOtherType(int, string);
};
MyOtherType m = {1, "b"};
// 如果相应的构造函数是 explicit 的，你就不能使用"= {}" 的形式。
MyOtherType m{"b"};
```

不要把 braced-init-list 赋值给一个 `auto` 局部变量。在列表中只有一个值的情况下，会引起误解。

```
auto d = {1.23};         // d 的类型是 initializer_list<double>。

auto d = double{1.23};   // 好 -- d 的类型是 double，而不是 initializer_list。
```

6.21 Lambda 表达式

不要使用 **lambda** 表达式，也不要使用相关的 **`std::function`** 或 **`std::bind`** 等工具。

6.21.1 定义：

Lambda 表达式是创建匿名对象的一种简洁方法。他们常用在传递函数作为参数时。如 `std::sort(v.begin(), v.end(), [](string x, string y) { return x[1] < y[1]; });`。Lambda 在 C++11 中引入，还有一组工具，用以和函数对象配合，如多态封装器 std::funtction。### 优点：- Lambda 比其它定义函数对象以传入 STL 算法的方法都要简单，可以提高可读性。- Lambda，std::function 和 std::bind 可以一起作为通用目的的回调机制使用；使得写以绑定的函数为参数的函数更简单。### 缺点：- Lambda 中的变量捕获很有技巧性，可能会成为新的悬空指针 BUG 滋生地。- 使用 lambda 可能失控；很长嵌套的匿名函数很难懂。### 结论：不要使用 lambda 表达式、std::function 或 std::bind`。

6.22 Boost

只使用 **Boost** 中被认可的库。

6.23 定义：

Boost 库集是一组受欢迎的经过同行评审的，免费开源的 C++ 库。

6.24 优点：

Boost 库普遍质量很高，可移植性好，并且填补了 C++ 标准库中一些重要的空白，如类型特性 (type traits)，更好的绑定器，以及更好的智能指针。它也实现了标准库的 TR1 扩展。

6.25 缺点：

一些 Boost 库提倡的编程实践可读性不好，如元编程和其它高级模板技巧，以及过度追求函数式编程。

6.26 结论：

为了对代码维护者和阅读者维持高度的可读性，我们只允许 Boost 库的一个经验证的子集。目前允许下述 Boost 库：

- Call Traits，来自 boost/call_traits.hpp
- Compressed Pair，来自 boost/compressed_pair.hpp

- The Boost Graph Library(BGL)，来自 boost/graph，除了序列化（adj_list_serialize.hpp）、并行/分布算法和数据结构（boost/graph/parallel/和 *boost/graph/distributed/*）。
- Property Map，来自 boost/property_map，除了并行/分布属性 map（boost/property_map/parallel/*）。
- Iterator中处理迭代器定义的部分：boost/ iterator/ iterator_adaptor.hpp,boost/ iterator/ iterator_facade.hpp, and boost/function_output_iterator.hpp
- Polygon中处理构造沃罗诺伊图（Voronoi Diagram）的部分，这部分不依赖 Polygon 其它部分：boost/ polygon/ voronoi_builder.hpp, boost/ polygon/ voronoi_diagram.hpp, and boost/ polygon/ voronoi_geometry_type.hpp
- Bitmap，来自 boost/bitmap
- Statistical Distributions and Functions，来自 boost/math/distributions

我们正在积极考虑添加其它 Boost 特性，所以这个列表以后会不断扩展。

下面的库也允许使用，但是不被鼓励，因为它们已经被 C++11 标准库取代了：

- Array，来自 boost/array.hpp：用`std::array`代替。
- Pointer Container，来自 boost/ptr_container：使用std::unique_ptr的容器代替。

6.27 C++11

在恰当的时候才使用 **C++11**（即先前的 **C++0x**）中的库和语言扩展。在你的工程中使用 **C++11** 特性之前先考虑一下对其它环境的可移植性问题。

6.27.1 定义：

C++11 是 ISO 的 C++ 标准的最新版本。对语言和库都作了重要改动。

6.27.2 优点：

C++11 已经成为官方标准，终将会被越来越多的 C++ 编译器支持。它标准化了一些我们已经在使用的通用 C++ 扩展，允许对一些操作速记，并且有一些性能和安全性提升。

6.27.3 缺点：

C++11 实际上比之前的标准要复杂得多（1300 页对 800 页），并且许多开发者都对其不太熟悉。一些特性对代码可读性和可维护性的长期影响尚不得而知。我们不知道相关工具什么时候才能一致地支持 C++11 那么多特性，特别是在强制使用旧版本工具的工程中。

和 Boost 库一样，一些 C++11 扩展所鼓励的编程实践也会降低可读性，如去掉了对阅读代码有帮助冗余检查（如类型名），还有鼓励模板元编译。其它复制了现有系统已有的功能的扩展，这可能导致混乱和转换成本。

6.27.4 结论

除非特别指明，C++11 中特性可以使用。除了本指南其它部分中描述的，下面的 C++11 特性不可以使用：

- 结尾返回类型的函数，如，使用 `auto foo() -> int;` 代替 `nt foo();`，因为要和许多已存在的函数声明保持一致。
- 编译期有理数（`<ratio>`），因为它被绑定到一个更加重量级的模板风格上。
- `<cfenv>` 和 `<fenv.h>` 头文件，因为许多编译器都不能可靠地支持那些特性。
- Lambda 表达式，或相关的 `std::function` 或 `std::bind` 工具。

7 命名

最重要的一致性原则就是管理命名。命名风格可以让我们无需查看声明就能立刻知道它的含义：是类型，变量，函数，常量还是宏等等。我们大脑的模式匹配引擎在很大程度上依赖这些命名规则。

命名规则有一定程度的随意性，但是我们认为在命名这件事上一致性比各自为政重要的多，所以不管你怎么想，规矩就是规矩。

7.1 通用命名规则

函数名，变量名和文件名应该是描述性的，并避免缩写。

尽量给一个名字合理的描述性。不要操心省一点行空间的事，代码可以让新的阅读者快速理解重要的多得多。不要用缩写，它会使你项目外的读者迷惑或不熟悉，也不要通过删除单词中的几个字母来缩写。

```
// 符合规定的命名
int price_count_reader;     // 没有缩写
int num_errors;             // "num" 是一个广为人知的用法
int num_dns_connections;    // 大多数人知道"DNS" 是什么意思

// 不符合规定的命名
int n;                      // 没有意义
int nerr;                   // 不清楚的缩写
int n_comp_conns;           // 不清楚的缩写
int wgc_connections;        // 只有你的团队知道是什么意思
int pc_reader;              // 许多东西都可以缩写成"pc"
int cstmr_id;               // 删除了中间的字母
```

7.2 文件命名

** 文件名应该都是小写字母。可以包含下划线（_）和破折号（-），用什么要遵循你项目的约定，如果没有标准，就用 ``_"。**

下面是可接受的文件名：

```
my_useful_class.cc
my-useful-class.cc
myusefulclass.cc
myusefulclass_test.cc // _unittest 和 _regtest 已被弃用
```

C++ 源文件后缀应是.cc，头文件后缀应为.h。

不要使用/usr/include 中已经存在的文件名，如 db.h。

通常应该让你的文件名更明确，如 http_server_logs.h 就比 logs.h 好。一个非常常规的用法就是用一对文件，比如名为 foo_bar.h 和 foo_bar.cc，来定义一个名为 FooBar 的类。

内联函数必须要.h 文件中。如果你的内联函数很短，就应该直接写在.h 文件中；如果很长，就应该写在另一个以 -inl.h 为后缀的文件中。对于有大量内联代码的类，可能有三个与之对应的文件：

```
url_table.h      // 类声明
url_table.cc     // 类定义
url_table-inl.h  // 大的内联函数
```

见 -inl.h 文件一节。

7.3 类型命名

类型名应该以大写字母开头，并且中间的每个单词都以大写字母开头，不使用下划线：**MyExcitingClass**，**MyExcitingEnum**。

所有的类型名（类，结构体，typedef，枚举等）都要使用相同的约定。类型名应该以大写字母开头，并且中间的每个单词都以大写字母开头。不需要使用下划线。如：

```
// 类和结构体
class UrlTable { ...
class UrlTableTester { ...
struct UrlTableProperties { ...

// typedef
typedef hash_map<UrlTableProperties *, string> PropertiesMap;

// 枚举
enum UrlTableErrors { ...
```

7.4 变量命名

变量名都是小写字母，单词使用下划线隔开。类成员以下划线结尾。如，**my_exciting_local_variable**，**my_exciting_member_variable_**

7.4.1 普通变量名

如：

```
string table_name;  // 好 - 使用下划线
string tablename;   // 好 - 所有的字母都是小写

string tableName;   // 不好 - 大小写混合
```

7.4.2 类数据成员

类数据成员（也叫实例变量或成员变量）和普通变量一样，是小写字母加可选的下划线，但是最后要以一个下划线结尾。

```
string table_name_;  // 好 - 下划线结尾
string tablename_;   // 好
```

7.4.3 结构体变量

结构体的数据成员应该和普通变量一样命名，结尾不需要和类成员一样的下划线。

```
struct UrlTableProperties {
  string name;
  int num_entries;
}
```

什么时候用结构体而不是用类请见结构体 VS 类一节。

7.4.4 全局变量

对全局变量命名没有特殊要求，它本来也极少使用，但如果你非要用一个全局变量，考虑使用 `g_` 或其它的前缀来使其很容易和局部变量区分开。

7.5 常量命名

用 **k** 跟着大小写混合的形式命名常量：**kDaysInAWeek**。

对所有编译期的常量，不管是局部的，全局的还是一个类中的，都遵循和其它变量稍微不同的命名约定。使用 `k` 跟着首字母大写的单词来命名。

```
const int kDaysInAWeek = 7;
```

7.6 函数命名

普通函数使用大小写混合模式命名：存取函数要和变量名匹配：**MyExcitingFunctions()**，**MyExcitingMethod()**，**my_exciting_member_variable()**，**set_my_exciting_member_variable()**。

7.6.1 普通函数

函数名应该以大写字母开头，中间的每一个单词都首字母大写。不需要下划线。

如果函数在碰到某些错误时会崩溃，你应该在函数名后面加上 `OrDie`。这些函数都是生产环境中使用的代码，且在常规操作时有可能会失败。

```
AddTableEntry()
DeleteUrl()
OpenFileOrDie()
```

7.6.2 存取函数

存取函数（**get/set** 函数）应该和它们操作的变量名匹配。下面摘录了一个类，这个类有一个名为 **num_entries_** 的成员变量。

```
class MyClass {
 public:
  ...
  int num_entries() const { return num_entries_; }
  void set_num_entries(int num_entries) { num_entries_ = num_entries; }

 private:
  int num_entries_;
};
```

非常短小的内联函数也可以使用小写字母。如，那些特别轻量的函数，轻的你在循环中调用都不会缓存其返回值，这时使用小写字母是可以接受的。

7.7 命名空间命名

命名空间名字都是小写的，基于工程名和目录结构：**google_awsome_project**。

关于命名空间的讨论详见命名空间。

7.8 枚举命名

枚举应该和常量或宏一样命名：或 **kEnumName** 或 **ENUM_NAME**。

每个枚举值应该优先以常量的形式命名。不过按宏方式命名也是可以接受的。枚举名，`UrlTableErrors`（以及 `AlternateUrlTableErrors`），是一个类型，因此使用大小写混合模式。

```
enum UrlTableErrors {
  kOK = 0,
  kErrorOutOfMemory,
  kErrorMalformedInput,
};
enum AlternateUrlTableErrors {
  OK = 0,
  OUT_OF_MEMORY = 1,
  MALFORMED_INPUT = 2,
};
```

2009 年 1 月之前，本指南还建议将枚举按宏一样命名。这会引起枚举值和宏之间的命名冲突，所以改为以常量风格命名。新代码应该尽量使用常量风格，但没有必要修改旧代码，除非它们产生了编译问题。

7.9 宏命名

通常都不需要定义宏,不是吗?如果你定义了宏,看起来应该是这样的:**MY_MACRO_THAT_SCARES_SMALL_CHILDREN**。

请查看对宏的描述：通常都不应该使用宏。但是如果确实需要宏，应该以全大写字母加下划线来命名。

```
#define ROUND(x) ...
#define PI_ROUNDED 3.0
```

7.10 命名规则的例外

如果你要命名的东西和已有的 **C/C++** 代码中的类似，你应该遵循已有的命名约定。

- `bigopen()`：函数名，以 `open()` 方式
- `uint`：typedef
- `bigpos`：结构体或类, 参照了 `pos` 的形式
- `sparse_hash_map`：STL 相似，遵循 STL 命名约定
- `LONGLONG_MAX`：常量，如同 `INT_MAX`

8 注释

尽管注释写起来很痛苦，但对代码可读性至关重要。下面规则描述了你应该在什么地方使用什么样的注释。但是要记住：尽管注释很重要，最好的代码却一定是可以自解释的。给类型和变量起一个合理的名字，要远远胜过用一个含糊的名字再用注释去解释。

注释是写给读者的：下一个需要理解你代码的贡献者。慷慨些吧，下一个人很可能就是你自己。

8.1 注释风格

** 用//和/* */都行 *，只要你自己保持一致。**

用//和/* */都行，但//更常用；你自己如何使用注释和注释风格要统一。

8.2 文件注释

文件开头应为版权许可证后面跟着本文件内容描述。

8.2.1 法律声明和作者信息

每一个文件都应该包含版权许可证。为你的项目选择一个合适的许可证（如，Apache 2.0，BSD，LGPL，GPL）。

如果你对一个有作者信息的文件做了重大的改动，考虑删除作者信息行。

Why?

8.2.2 文件内容

每一个文件头部都要有一段描述其内容的注释。

通常.h 文件应该描述它声明的类，以及类作用和使用方法的简要说明。.cc 文件应该包含实现细节或算法技巧描述等更多信息。如果你认为实现细节或算法讨论会对.h 读者有用，就把它们放在.h 文件中，只是别忘了在.cc 文件中要提醒一下它们在.h 文件。

不要在.h 和.cc 文件之间复制注释。复制注释就失去注释的真正意义了。

8.3 类注释

每一个类定义都要附带一个注释来描述其功能和用法。

```
// GargantuanTable 内容的迭代器。使用示例：
//    GargantuanTableIterator* iter = table->NewIterator();
//    for (iter->Seek("foo"); !iter->done(); iter->Next()) {
//      process(iter->key(), iter->value());
//    }
//    delete iter;
class GargantuanTableIterator {
  ...
};
```

如果文件头部已经有了类的描述，直接来一句 ``完整描述见文件头" 也没问题，但一定要确保有此类注释。

如果有同步相关的假设，就要写下来。一个类对象是否可以被多线程访问，多线程下规则和常量的使用在文档化时要格外注意。

8.4 函数注释

函数需要声明注释；函数定义处的注释描述操作。

8.4.1 函数声明

每个函数的前面都要有注释来说明其功能和用法。这些注释应该是描述性的（``Opens the file''）而不是命令式的（``Open the file''）；注释是描述函数用的，而不是告诉函数干什么的。通常，这些注释都不会描述函数如何执行任务，这应该是函数定义处的注释要做的事。

函数声明注释中要提及的内容：

- 输入输出。
- 对于类成员函数：对象是否会在调用它之外记住引用参数，它是否会释放这些引用参数。
- 如果函数分配了内存，调用者必须负责释放
- 参数可否为空指针
- 使用时有没有隐含的性能开销
- 是否可以重入。其同步假设是什么？

下面是一个例子：

```
// 返回这个表的一个迭代器。使用完后需要用户来删除这个迭代器。
// 迭代器指向的 GargantuanTable 对象被删除后就不能再使用这个迭代器了。
//
// 此迭代器初始指向表头
//
// 这个方法等价于：
//     Iterator* iter = table->NewIterator();
//     iter->Seek("");
//     return iter;
// 如果你拿到这个迭代器会立刻查找另一个位置，用 NewIterator() 会更快，
// 并可以避免额外的查找操作。
Iterator* GetIterator() const;
```

然而，没有必要罗里罗嗦地去做些显而易见的说明。注意下面的例子就没有必要说 ``否则返回 `false`''，因为这个已经隐含了。

```
// 如果表满了就返回 true
bool IsTableFull();
```

当注释构造函数和析构函数时，要清楚读者是明白构造函数和析构函数的，所以类似 ``销毁此对象" 这种注释是没有用的。需要说明是构造函数用参数来干什么（如，是否会持有指针），以及析构函数做了什么样的清理工作。析构函数通常都不需要头文件注释。

8.4.2 函数定义

函数中使用的任何技巧都应该在注释中说明。如，在一个函数定义注释中，你可能会描述你使用的编码技巧，给出大体的实现步骤，以及你为什么要这样而不是那样实现此函数。你也会提及为什么前半段代码需要锁，而后半段不需要。

注意不要只是简单重复头文件或其它什么地方的声明注释。简要概括一下函数是可以的，但着重是要注释如何实现。

8.5 变量注释

通常变量名应该足以说明变量的用途。在一些特定情况下，才需要多一点注释。

8.5.1 类数据成员

每一个类数据成员（又称实例变量或成员变量）都要注释其用途。变量是否可以持有特定含义的哨兵值，如空指针或 -1，也要说明。举个例子：

```
private:
 // 跟踪表的项数。用来保证不越界。
 // -1 表示我们还不知道表的项数。
 int num_total_entries_;
```

8.5.2 全局变量

和数据成员一样，全局变量也要用注释来说明其含义和用法。如：

```
// 本次回归测试中总共跑的测试用例数
const int kNumTestCases = 6;
```

8.6 实现注释

在你的实现中，应该注释技巧、不明显的、有趣的或重要的部分。

8.6.1 类函数成员

技巧和复杂代码块前需要注释。如：

```
// 将结果除 2，注意 x 保存进位
for (int i = 0; i < result->size(); i++) {
  x = (x << 8) + (*result)[i];
  (*result)[i] = x >> 1;
  x &= 1;
}
```

8.6.2 行注释

不明确的行尾部也要添加注释。这些注释和代码之间要有 2 个空格。如：

```
// If we have enough memory, mmap the data portion too.
mmap_budget = max<int64>(0, mmap_budget - index_->length());
if (mmap_budget >= data_size_ && !MmapData(mmap_chunk_bytes, mlock))
  return;  // Error already logged.
```

注意上面既有表述代码作用的注释，也有注释提醒函数返回时已经记录了日志。

如果在连续的行上都有注释，将它们对齐更可读：

```
DoSomething();                    // Comment here so the comments line up.
DoSomethingElseThatIsLonger();    // Comment here so there are two spaces between
                                  // the code and the comment.
{ // One space before comment when opening a new scope is allowed,
  // thus the comment lines up with the following comments and code.
  DoSomethingElse();  // Two spaces before line comments normally.
}
DoSomething(); /* For trailing block comments, one space is fine. */
```

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

当传给函数一个空指针，布尔值，或一个字面数值时，应该考虑注释一下它们是什么意思，或用常量使你的代码可以自说明。如，对比下面两段代码：

```
bool success = CalculateSomething(interesting_value,
                                  10,
                                  false,
                                  NULL);  // What are these arguments??
```

和

```
bool success = CalculateSomething(interesting_value,
                                  10,     // Default base value.
                                  false,  // Not the first time we're calling this.
                                  NULL);  // No callback.
```

抑或使用可以自解释的常量也行:

```
const int kDefaultBaseValue = 10;
const bool kFirstTimeCalling = false;
Callback *null_callback = NULL;
bool success = CalculateSomething(interesting_value,
                                  kDefaultBaseValue,
                                  kFirstTimeCalling,
                                  null_callback);
```

8.6.4 不允许

注意永远都不要对代码进行解释。假设读代码的人即便不知道你要做什么，他的 C++ 水平也要比你高：

```
// Now go through the b array and make sure that if i occurs,
// the next element is i+1.
...        // Geez.  What a useless comment.
```

8.7 标点，拼写和语法

要在标点、拼写和语法上心思；写得好的注释更易读。

注释应该和叙事文本一样易读，要有正确的大小写和标点符号。通常完整的句子要比片段更易读。短注释，如行尾注释，有时可以不那么正规，但你自己也要保持一致的风格。

虽然让代码评审者指出你应该用分号时使用了逗号很让人不爽，但让源代码保持一个高层次的清晰性和可读性是非常重要的。合理的标点、拼写和语法对此会有所帮助。

8.8 TODO 注释

对临时代码、短期解决方案以及可用但不够完美的代码使用 **TODO** 注释。

TODO 应该包含全大写的字符串 TODO，后面跟着可以提供这个问题来龙去脉的人的大名、邮箱或其它标识。后面再一个可选的冒号。这么做的主要目的是为了有一个一致的 TODO 格式，可以查找能为该请求提供更多细节的人。TODO 不是用来注释某人以后会修正这个问题的。所以，当添加一条 TODO 时，总是应该写上你自己的名字。

```
// TODO(kl@gmail.com): Use a "*" here for concatenation operator.
// TODO(Zeke) change this to use relations.
```

当你的 TODO 是类似 ``未来会如何如何" 这种形式时，要确保包含确切的日期（``2005 年 11 月修正"）或特定事件（``当所有客户都能处理 XML 文件时，就移除所有这些代码"）。

8.9 弃用声明注释

对弃用的接口使用 **DEPRECATED** 注释进行说明。

你可以用包含全大写的单词 DEPRECATED 的注释来标记一个已经弃用的接口。这个注释或者放在接口的声明之前或者放在同一行。

在单词 DEPRECATED 后面的括号里写上你的大名，邮箱，或其它身份标识。

一个弃用注释必须包含简单清楚的用法说明，来帮助人们修改他们的调用点。C++ 中，你可以把弃用函数声明成一个内联函数，并在其中调用新的接口。

把一个接口标记成 DEPRECATED 并不能自动修改调用点。如果你真的需要调用者停止使用弃用的设施，你应该自己去修改调用点或纠集一帮人来帮你干这个事。

新代码不应该调用弃用的接口，而要使用新接口。如果你看不懂用法说明，就找创建这个弃用的人，让他们教你使用新接口。

9 格式

编码风格和格式可以很随意，但是在一个工程中遵循相同的风格要容易得多。个体可以不同意格式规定中的所有项，一些规则可能需要时间适应，但是所有贡献者使用相同的代码风格很重要，这使他们自己也可以更容易地阅读和理解每个人的代码。

为了帮助你以正确格式编码，我们已经创建了一个 emacs 配置文件。

9.1 行长度

你代码中的每一行文本长度应最多不能超过 **80** 个字符。

我们知道这条规则有争议，但是许多已有代码都在遵循它，我们觉得保持这种一致性很重要。

9.1.1 优点：

本条规则的拥护者认为强迫他们放大窗口是一种冒犯，同时也没有必要。一些人同时并排开了几个编程窗口，因此根本没有多余空间来拉伸他们的窗口。人们设置他们的工作环境时会假设一个窗口的最大宽度，80 列已经成为传统的标准，为什么要改变？

9.1.2 缺点：

支持改变的人认为更宽的行可以使代码更可读。80 列限制是顽固地倒退回上世纪 60 年代大型机的时代。现代的显示器拥有更宽的屏幕，可以轻松显示更长的行。

9.1.3 结论：

80 个字符是最大值。

- 例外：如果一个注释行包含了示例命令或一个超过 80 字符的 URL，那么这一行可以超过 80 个字符以方便拷贝粘贴。
- 例外：当 `#include` 语句后面的路径长度超过 80 列宽时。要尽量避免这种情况。
- 例外：不需要关心头文件保护超过长度限制。

9.2 非 ASCII 字符

非 **ASCII** 字符极少需要，用也要是 **UTF-8** 编码格式。

你不应该把用户界面文本硬编码到代码中，即使是英语也不行，所以需要使用非 ASCII 字符的情况非常之少。但有些特殊情况适合包含此类单词。例如，如果你的代码要解析外文数据文件，硬编码文件中一些非 ASCII 字符串作为分隔符是合理的。更常见的，单元测试代码（不需要本地化）可能包含非 ASCII 字符串。在这些情况下你应该使用 UTF-8 编码，因为除了 ASCII 编码外，大多数工具都能理解 UTF-8 编码。

Hex 编码也可以，在有助于可读性的情况下尤为鼓励，如，`"\xFE\xBB\xBF"`，或更简单的，`u8"\uFEFF"` 是一个零宽度、无间断的 UNICODE 空白字符，如果在源代码中直接使用 UTF-8 就是不可见的。

使用 `u8` 来确保一个包含 `\uXXXX` 转义序列的字符串字面值会以 UTF-8 编码。不要在包含 UTF-8 编码的非 ASCII 字符的字符串中使用它，因为如果编译器不把源文件解释成 UTF-8，这会产生错误的输出。

> 不太明白

不要使用 C++11 中的 `char16_t` 和 `char32_t` 类型，因为它们是用于非 UTF-8 文本的。同样的原因，你也不应该使用 `wchar_t`（除非你在写和 Windows API 交互的代码，其中大量使用 `wchar_t`）。

9.3 空格和 Tab

只使用空格，每次缩进 **2** 个空格。

我们使用空格缩进。不要在你的代码中使用制表符。你应该设置你的编辑器来把 tab 键变成空格。

9.4 函数声明和定义

返回值类型和函数名放在同一行上，参数也尽量放在同一行上。

函数看上去应该这样：

```
ReturnType ClassName::FunctionName(Type par_name1, Type par_name2) {
  DoSomething();
  ...
}
```

如果你的一行太长放不下所有参数：

```
ReturnType ClassName::ReallyLongFunctionName(Type par_name1, Type par_name2,
                                             Type par_name3) {
  DoSomething();
  ...
}
```

抑或是连一个参数都放不下：

```
ReturnType LongClassName::ReallyReallyReallyLongFunctionName(
    Type par_name1,  // 4 个空格缩进
    Type par_name2,
    Type par_name3) {
  DoSomething();  // 2 个空格缩进
  ...
}
```

需要指明几点：

- 如果不能把函数返回类型和函数名放在一行上，可以在中间加换行。
- 如果在函数的返回类型之后换行，就不要缩进了。
- 左圆括号也总是和函数名在同一行上。
- 左括号和函数名之间没有空格。
- 括号和参数之间没有空格。
- 左大括号总是和最后一个参数在同一行上。
- 右大括号或者单独一行，或者（如果不违背其它规则）和左大括号在同一行上。
- 右小括号和左大括号之间要有一个空格。
- 所有参数都要具名，声明和实现时都要有标识名。
- 所有参数都应该尽量对齐。
- 默认缩进是 2 个空格。
- 换行的参数使用 4 字节缩进。

如果有参数没有使用，在函数定义时把这个变量名注释出来：

```
// 接口中总是使用具名参数
class Shape {
 public:
  virtual void Rotate(double radians) = 0;
}

// 声明中总是使用具名参数
class Circle : public Shape {
 public:
  virtual void Rotate(double radians);
}

// 定义时注释未使用的参数名
void Circle::Rotate(double /*radians*/) {}

// 不好 - 如果有人以后想实现这个函数，这个变量的含义就不清楚
void Circle::Rotate(double) {}
```

9.5 函数调用

尽量放在同一行，否则换行圆括号中的实参。

函数调用形式如下：

```
bool retval = DoSomething(argument1, argument2, argument3);
```

如果参数不能放在同一行上，就应该断到多行，后面的每一行都和第一个参数对齐。不要在左括号后和右括号前不要有空格。

```
bool retval = DoSomething(averyveryveryverylongargument1,
                          argument2, argument3);
```

如果函数有太多参数，考虑每行一个来使代码更易读：

```
bool retval = DoSomething(argument1,
                          argument2,
                          argument3,
                          argument4);
```

参数也可以都放在函数名下面的行上，一行一个：

```
if (...) {
  ...
  ...
  if (...) {
```

```
    DoSomething(
        argument1,  // 4 space indent
        argument2,
        argument3,
        argument4);
  }
```

特别是当函数签名太长放在同一行时会超过一行的最大长度，更要如此。

9.6 大括号初始化列表

和函数调用一样格式化大括号初始化列表。

如果列表跟在一个名字后（如一个类型或一个变量名），就把 {} 看作函数调用中的小括号一样格式化。如果没有名字，假想一个 0 长度的名字。

尽量把所有的东西都放到一行上。如果不能放到一行上，左大括号应该是其所在行的最后一个字符，且右大括号应该是其所在行的第一个字符。

```
// 单行大括号列表示例。
return {foo, bar};
functioncall({foo, bar});
pair<int, int> p{foo, bar};

// 需要换行时。
SomeFunction(
    {"assume a zero-length name before {"},
    some_other_function_parameter);
SomeType variable{
    some, other, values,
    {"assume a zero-length name before {"},
    SomeOtherType{
        "Very long string requiring the surrounding breaks.",
        some, other values},
    SomeOtherType{"Slightly shorter string",
                  some, other, values}};
SomeType variable{
    "This is too long to fit all in one line"};
MyType m = {  // Here, you could also break before {.
    superlongvariablename1,
    superlongvariablename2,
    {short, interior, list},
    {interiorwrappinglist,
     interiorwrappinglist2}};
```

9.7 条件语句

尽量括号中不要有空格。**else** 关键字另起一行。

一个基本的条件语句有两种可接受的形式。一种在小括号和条件之间有空格，另一种没有。

最常用的形式是没有空格的。另一种也可以，但是要保持一致。如果你在修改一个文件，使用已有的格式。对于新代码，使用同一目录或同一工程中的形式。如果不确定并且没有个人倾向，就不要用空格。

```
if (condition) {  // 小括号里没有空格
  ...  // 2 个空格缩进
} else if (...) {  // 和右大括号在同一行的 else 语句。
  ...
} else {
  ...
}
```

如果你选择在小括号中添加空格：

```
if ( condition ) {  // 小括号中有空格 - 少用
  ...  // 2 个空格缩进
} else {  // 和右大括号在同一行的 else 语句。
  ...
}
```

注意无论何种形式，你都必须在 `if` 和左小括号之间加空格。如果有大括号，右小括号和左大括号之间也要有空格。

```
if(condition)      // 不好 - if 后面缺了空格。
if (condition){    // 不好 - {后面缺了空格。
if(condition){     // 更不好

if (condition) {  // 好 - if 后和 {前都有合适的空格。
```

如果可以加强可读性，短的条件语句可以放在一行。只有在代码行非常精简并且没有使用 `else` 语句时才能这样。

```
if (x == kFoo) return new Foo();
if (x == kBar) return new Bar();
```

当 `if` 语句有 `else` 子句时，这是不允许的：

```
// 不允许 - IF 语句在有 ELSE 时还放在同一行上
if (x) DoThis();
else DoThat();
```

通常，单行语句不需要花括号，但是如果你喜欢也可以使用；有复杂条件或语句的条件或循环语句使用花括号更易读。一些工程要求 if 必必须要有对应的大括号。

```
if (condition)
  DoSomething();  // 2 个空格缩进。

if (condition) {
  DoSomething();  // 2 个空格缩进。
}
```

然而，如果 if-else 语句的某一部分使用了花括号，其它的部分也必须要使用：

```
// 不允许 - IF 使用了花括号而 ELSE 没有
if (condition) {
  foo;
} else
  bar;

// 不允许 - ELSE 使用了花括号而 IF 没有
if (condition)
  foo;
else {
  bar;
}

// 因为一部分使用了花括号，所以 IF 和 ELSE 都需要使用
if (condition) {
  foo;
} else {
  bar;
}
```

9.8 循环和 Switch 语句

switch 语句可以使用大括号分块。要注释 **case** 之间重要的失败。空循环体应该用 **{}** 或 **continue**。

switch 语句中的 case 块可以使用也可以不使用花括号，这取决于你的喜好。如果要使用花括号，需要像下面的示例那样放置。

如果不是以枚举值为条件，switch 语句应该有一个 default 匹配（如果使用枚举值，对没有处理的值编译器会有警告）。如果 default 匹配永远都不应该发生，简单使用一个 assert 即可：

```
switch (var) {
  case 0: {  // 2 个空格缩进
    ...      // 4 个空格缩进
    break;
```

```
  }
  case 1: {
    ...
    break;
  }
  default: {
    assert(false);
  }
}
```

空循环体应该用 {} 或 continue，而不能只一个单独的分号。

```
while (condition) {
  // 重复测试直到返回 fasle。
}
for (int i = 0; i < kSomeNumber; ++i) {}  // 好 - 空循环体。
while (condition) continue;  // 好 - continue 表明无逻辑。

while (condition);  // 不好 - 看起来像 do/while 循环的一部分。
```

9.9 指针和引用表达式

点和箭头周围都不要有空格。指针操作符后面也不要有空格。

下面都是正确格式的指针和引用示例：

```
x = *p;
p = &x;
x = r.y;
x = r->y;
```

注意：

- 访问成员时的点号和箭头周围都没有空格。
- 指针操作符 * 或 & 后面没有空格。

当声明指针变量或参数时，你让星号靠近类型或靠近变量名都可以：

```
// 下面这样可以，星号前面加空格。
char *c;
const string &str;

// 下面这样也可，星号后面加空格。
char* c;     // 但是要记住多个变量的时候："char* c, *d, *e, ...;"!
const string& str;
```

```
char * c;  // 不好 - * 前后都有空格
const string & str;  // 不好 - & 前后都有空格
```

在一个文件中使用的形式要一致，所以修改文件时，要使用文件中已用的风格。

9.10 布尔表达式

当你的布尔表达式超过标准行长度时，如何换行要保持一致。

下面的例子中，逻辑与操作符总是在行尾：

```
if (this_one_thing > this_other_thing &&
    a_third_thing == a_fourth_thing &&
    yet_another && last_one) {
  ...
}
```

注意在这个例子中，两个逻辑与操作符 `&&` 都在行尾。这在 Google 的代码是很常用，尽管把操作符放在行首的断行方式也是允许的。放心大胆地合理使用小括号，因为如果使用得当它们会极大增加可读性。同时也要注意，总是使用符号操作符，如 `&&` 和 `~`，而不要使用文字操作符，如 `and` 和 `cmpl`。

9.11 返回值

不要徒劳地用小括号包围起 **`return`** 表达式。

只有当你在 `x = expr` 中也要使用括号时，才需要在 `return expr` 是使用小括号。

```
return result;                  // 简单情况不用括号。
return (some_long_condition &&  // 括号使用正确，增加了复杂表达式的可读性
        another_condition);

return (value);                // 不好！ 不是 var = (value) 的形式；
return(result);                // 不好！ return 不是函数！
```

9.12 变量和数组初始化

用 **`=`**，**`()`** 或 **`{}`** 均可。

你可以在 `=`，`()` 和 `{}` 之间选择，下面都是正确的：

```
int x = 3;
int x(3);
int x{3};
string name = "Some Name";
string name("Some Name");
string name{"Some Name"};
```

在有接收 initializer_list 的构造函数的类型上使用 {} 要小心，{} 语法会尽可能优先选择 initializer_list 构造函数。要使用其它构造函数，应使用 ()。

```
vector<int> v(100, 1);  // A vector of 100 1s.
vector<int> v{100, 1};  // A vector of 100, 1.
```

大括号形式也可以阻止整数类型窄化转型 (narrowing conversion)，这可以防止一些编程错误。

```
int pi(3.14);  // OK -- pi == 3.
int pi{3.14};  // 编译器错误：  窄化转型
```

9.13 预处理指令

井号开头的预处理指令应该在行首。

即使预处理指令在缩进的代码段中，也要从行首开始。

```
// 好 - 指令在行首
  if (lopsided_score) {
#if DISASTER_PENDING      // 正确 -- 从行首开始
    DropEverything();
# if NOTIFY               // 可以但不要求 # 后有空格
    NotifyClient();
# endif
#endif
    BackToNormal();
  }

// 不好 - 缩进指令
  if (lopsided_score) {
    #if DISASTER_PENDING  // 错误！ “#if”应该在行首
    DropEverything();
    #endif                // 错误！不要缩进“#endif”
    BackToNormal();
  }
```

9.14 类格式

代码段顺序为 **public**，**protected** 和 **private**，各缩进一个空格。

类声明的基本形式（不包含注释，参见类注释相关讨论）是：

```
class MyClass : public OtherClass {
 public:      // 注意一个空格缩进！
  MyClass();  // 普通的 2 空格缩进。
```

```
  explicit MyClass(int var);
  ~MyClass() {}

  void SomeFunction();
  void SomeFunctionThatDoesNothing() {
  }

  void set_some_var(int var) { some_var_ = var; }
  int some_var() const { return some_var_; }

 private:
  bool SomeInternalFunction();

  int some_var_;
  int some_other_var_;
  DISALLOW_COPY_AND_ASSIGN(MyClass);
};
```

注意事项：

- 任何基类名都应该和子类在同一行上，但受限于 80 列限制。
- public:、protected: 和 private: 关键字应缩进一个空格。
- 上面的关键字，除了第一个出现的，都应该前面加一空行。这条规则在较小的类中是可选的。
- 上面的关键字后面不要留空行。
- 先是 public 段，接着是 protected 段，最后是 private 段。
- 每一段内部的声明顺序参见声明顺序一节。

9.15 构造函数初始化列表

构造函数初始化列表可以在同一行上，也可以分行，后面的行都缩进 **4** 个空格。

初始化列表有两种可接受的形式：

```
// 可以放在同一行上：
MyClass::MyClass(int var) : some_var_(var), some_other_var_(var + 1) {}
```

或

```
// 需要多行，缩进 4 个空格，算上第一行的冒号
MyClass::MyClass(int var)
```

```
    : some_var_(var),             // 4 空格缩进
      some_other_var_(var + 1) {  // 和上一行对齐
  ...
  DoSomething();
  ...
}
```

9.16 命名空间格式化

命名空间内容不缩进。

命名空间不增加缩进层次。如:

```
namespace {

void foo() {  // 正确。命名空间里不需要额外的缩进。
  ...
}

}  // namespace
```

命名空间不需要缩进:

```
namespace {

  // 错误。不应该有缩进。
  void foo() {
    ...
  }

}  // namespace
```

当声明了嵌套命名空间时，每个一行:

```
namespace foo {
namespace bar {
```

9.17 水平空白

水平空白取决于位置。不要在行尾添加空白。

9.17.1 普通

```
void f(bool b) {  // 左大括号前面总需要空格。
  ...
int i = 0;  // 分号前面通常没有空格。
int x[] = { 0 };  // 大括号初始化列表内部的空格是可选的。
int x[] = {0};     // 但如果要用，就两边都用！
// 继承和初始化列表中的冒号周围要有空格。
class Foo : public Bar {
 public:
  // 对于内联函数实现，在大括号和实现代码之间加空格。
  Foo(int b) : Bar(), baz_(b) {}  // 空的大括号内不需要空格。
  void Reset() { baz_ = 0; }  // 用空格分隔大括号和实现代码。
  ...
```

行尾添加空格和给其它编辑代码的人造成额外的负担，当他们合并时，会删除已存在的空白。所以，不要引入行尾空白。编辑时删除行尾的空白，或通过单独的清理操作删除（当然要在没有其它人正在使用此文件时）。

9.17.2 循环和条件语句

```
if (b) {          // 条件和循环中关键字后面有空格。
} else {          // else 周围有空格
}
while (test) {}   // 小括号内通常没有空格。
switch (i) {
for (int i = 0; i < 5; ++i) {
switch ( i ) {    // 循环和条件的小括号中可以有空格，但不常用，且要自己保持一致。
if ( test ) {
for ( int i = 0; i < 5; ++i ) {
for ( ; i < 5 ; ++i) {  // for 循环中，分号之后一定要有空格，前面也可以有
  ...
for (auto x : counts) {  // 基于范围的 for 循环冒号前后总需要空格
  ...
}
switch (i) {
  case 1:         // switch 中 case 子句前的冒号不需要空格。
    ...
  case 2: break;  // 如果冒号后有代码，就需要加空格。
```

9.17.3 操作符

```
x = 0;              // 赋值操作符周围需要空格。
x = -5;             // 一元操作符和其参数之间不需要空格。
```

```
++x;
if (x && !y)
  ...
v = w * x + y / z;   // 二元操作符周围通常都有空格，
v = w*x + y/z;       // 但删除因子周围的空格也可以。
v = w * (x + z);     // 小括号内不需要空格。
```

9.17.4 模板和类型转换

```
vector<string> x;              // 尖括号内部（< 和>），< 之前或> (之间不需要空格。
y = static_cast<char*>(x);
vector<char *> x;              // 类型和指针符号间可以有空格，但要保持一致。
set<list<string>> x;           // C++11 代码中允许。
set<list<string> > x;          // C++03 要求> > 之间有一空格。
set< list<string> > x;         // 你也可以在< < 间使用空格来保护对称性。
```

9.18 垂直空白

尽可能少用垂直空白。

这更像是原则而不是规则：不要随便使用空行。特别是在函数之间不要添加起过一两个空行，函数开始不要有空行，也不要以空行结束，并且在函数体中使用空行也要节制。

基本原则是：一屏能显示的代码越多越好，这就越容易跟踪和理解程序的控制流。当然，过于密集和过于松散的代码可读性同样不好，这你要自己判断。但是，通常都是垂直空白越少越好。

一些经验法则可以帮助确定何时空行是有用的：

- 函数内头尾的空行对可读性几乎没有帮助。
- `if-else` 代码链中的空行有助于可读性。

10 规则特例

上面的编码约定都是强制性的。但是和所有好的规则一样，有时候也会有例外，我们会在这里讨论这样的情况。

10.1 现有的不合规范代码

处理不符合本指南规则的代码时，你可以变通一下。

如果你发现你正在修改的代码使用的规则不来自本指南，那么可以有一些变通以和原代码中的风格保持一致。如果你不知道怎么做，就去问原作者或现在对代码负责的人。记住，一致性也包含本地一致性。

10.2 Windows 代码

Windows 程序员已经有了一组编码约定，主要演变自 **Windows** 头文件和其它微软代码。我们希望任何人都能轻松看懂我们的代码，所以我们对在任何平台上写 **C++** 的人有单独的一组规则。

有必要重申几点，如果你习惯 Windows 风格，可能忘记的规则：

- 不要使用匈牙利命名法（如把一个整数命名为 `iNum`）。使用 Google 的命名约定，包括源代码文件应使用.cc 扩展名。

- Windows 定义了许多原生类型的同义词，如 `DWORD`，`HANDLE` 等。在调用 Windows API 函数时使用这些类型是可接受的，并且是被鼓励的。即使如此，也要尽可以接近原生的 C++ 类型。如，使用 `const TCHAR *` 来代替 `LPCTSTR`。

- 使用微软的 Visual C++ 编译时，把编译器警告级别设为 3 或更高，并把所有警告当成错误。

- 不要使用 `#pragma once`；应该使用标准的 Google 头文件保护，其中的路径应该是相对于工程顶层目录的相对的路径。

- 事实上，任何非标准扩展都不要使用，像 `#pragma` 和 `__declspec`，除非你确实非用不可。使用 `__declspec(dllimport)` 和 `__declspec(dllexport)` 是允许的;然而,你必需通过 `DLLIMPORT` 和 `DLLEXPORT` 这样宏来使用它们，以使其它人分享这些代码时可以轻易禁用这些扩展。

不过，我们还是有几条规则在 Windows 上偶尔会被打破：

- 通常我们都禁止使用多重实现继承；然而，在使用 COM 和一些 ATL/WTL 类时，这却是必须的。你可以使用多重实现继承来实现 COM 或 ATL/WTL 类和接口。

- 尽管你不应该在你自己的代码中使用异常，但异常在 ATL 和一些 STL 实现（包括 Visual C++ 中带的那份）中却被广泛使用。使用 ATL 时，你应该定义 `_ATL_NO_EXCEPTIONS` 宏来禁用异常。你也要调查一下你的 STL 版本能否也禁用异常，如果不能，在编译器中打开异常也没问题。（注意这只是为了编译 STL。你还是不应该用异常处理你自己的代码。）

- 使用预编译头文件的通常方式是在每个源文件的头部都包含一个头文件，文件名一般是 StdAfx.h 或 precompile.h。为了使你的代码更容易和其它工程分享，避免显式包含这个文件（除了在 precopile.cc 中），应该使用/FI 编译选项来自动包含它。

- 资源头文件，通常名为 resource.h 并且只包含一些宏，不需要遵守本指南的风格。

11 结束语

运用常识，并 保持一致性。

如果你正在写代码，停几分钟，看看你周围的代码并确定其风格。如果它们在 `if` 语句周围使用空格，你也要这么做。如果它们的注释用星号围成的盒子包围，你也要如此。

使用风格指南的核心是使用通用的词汇来让人们可以准确知道你在说什么，而不是聚焦于如何表达上。我们列出公共的风格规则就是要让人知道这些词汇。但本地风格也很重要。如果你向一个文件中添加的代码和周围的代码很不搭，这种中断会打乱读者的节奏，要尽量避免。

好了，关于代码风格已经写得够多了；代码本身要有趣得多。尽情享受吧！