Google C++ Style Guide 摘要

9sheng

spliun60@126.com

2014-03-13

任何一个傻瓜都能写出计算机可以理解的代码。唯有写出人类容易理解的代码，才是优秀的程序员。

— *Martin Flower*

任何一个傻瓜都能写出计算机可以理解的代码。唯有写出人类容易理解的代码，才是优秀的程序员。

— *Martin Flower*

据说一见钟情都是以貌取人，你会对代码一见钟情么？

这是什么，不是什么？

这是什么，不是什么？

- 是笔记，不是原创

这是什么，不是什么？

- 是笔记，不是原创
- 是理念，不是规范

这是什么，不是什么？

- 是笔记，不是原创
- 是理念，不是规范
- 是实战，不是理论

这是什么，不是什么？

- 是笔记，不是原创
- 是理念，不是规范
- 是实战，不是理论
- 是大杂烩，不限于 google 规范

这是什么，不是什么？

- 是笔记，不是原创
- 是理念，不是规范
- 是实战，不是理论
- 是大杂烩，不限于 google 规范
- 是平沙落雁式，不是葵花宝典

避免多重引用

```
<PROJECT>_<PATH>_<FILE>_H_
```

避免多重引用

```
<PROJECT>_<PATH>_<FILE>_H_
```

头文件引用方式

```
// google-awesome-project/src/base/logging.h
#include "logging.h"        // error
#include "../logging.h"     // error
#include "base/logging.h" // ok
```

避免多重引用

```
<PROJECT>_<PATH>_<FILE>_H_
```

头文件引用方式

```
// google-awesome-project/src/base/logging.h
#include "logging.h"       // error
#include "../logging.h"    // error
#include "base/logging.h" // ok
```

头文件的引用顺序，当前文件：dir2/foo2.cc

```
// dir2/foo2.h
// C 系统文件
// C++ 系统文件
// 其他库头文件
// 本项目内头文件
```

减少编译依赖

- 尽量少在 .h 中 include 其他 .h

减少编译依赖

- 尽量少在 .h 中 include 其他 .h

```
#include "Foo.h" // 引用头文件
class Foo;       // 前置声明
```

减少编译依赖

- 尽量少在 .h 中 include 其他 .h

```
#include "Foo.h" // 引用头文件
class Foo;       // 前置声明
```

依赖关系示例

```
Bar.h: #include "Foo.h"

Foo.h -> Foo.cpp
      -> Bar.h -> Bar.cpp
               -> Test1.cpp
               -> Test2.cpp
```

```
Bar.h: class Foo;

Foo.h -> Foo.cpp
      -> Bar.cpp

Bar.h Test1.cpp Test2.cpp
```

前置声明 VS. 包含头文件

- 前置声明
  - 数据成员类型为 `Foo*` 或 `Foo&`
  - 函数参数、返回值类型为 `Foo`
  - 静态数据成员声明为 `Foo`
- 必须包含头文件
  - 如果类是 `Foo` 的子类
  - 如果 `Foo` 为模板类
- 行为未定义
  - Foo 类重载了 `operator&`

前置声明 VS. 包含头文件

- 前置声明
  - 数据成员类型为 `Foo*` 或 `Foo&`
  - 函数参数、返回值类型为 `Foo`
  - 静态数据成员声明为 `Foo`
- 必须包含头文件
  - 如果类是 `Foo` 的子类
  - 如果 `Foo` 为模板类
- 行为未定义
  - Foo 类重载了 `operator&`

WHY?

命名空间

- 不能在. h 文件中使用匿名空间
- 在.cc 文件中，提倡使用匿名空间（不提倡使用 static）
- 避免直接使用 using 提示符污染命名空间
- 在.cc 文件、.h 文件的方法或类中，可以使用 using

命名空间

- 不能在. h 文件中使用匿名空间
- 在.cc 文件中，提倡使用匿名空间（不提倡使用 static）
- 避免直接使用 using 提示符污染命名空间
- 在.cc 文件、.h 文件的方法或类中，可以使用 using

匿名空间

```
// Bar.cpp
namespce {
  char c;
  int i;
  double d;
}
```

```
// Bar.cpp
namespace __UNIQUE_NAME_ {
  char c;
  int i;
  double d;
}
using namespace __UNIQUE_NAME_;
```

嵌套类

- 符合局部使用原则，尽量不要使用 public 嵌套类
- 原因：只有在被嵌套类的定义中才能前置声明嵌套类
- 例外：嵌套类是接口的一部分

嵌套类

- 符合局部使用原则，尽量不要使用 public 嵌套类
- 原因：只有在被嵌套类的定义中才能前置声明嵌套类
- 例外：嵌套类是接口的一部分

局部变量

- 声明即初始化（RAII）
- 尽可能延迟声明局部变量

嵌套类

- 符合局部使用原则，尽量不要使用 public 嵌套类
- 原因：只有在被嵌套类的定义中才能前置声明嵌套类
- 例外：嵌套类是接口的一部分

局部变量

- 声明即初始化（RAII）
- 尽可能延迟声明局部变量

```
// f 定义应放在 for 循环之外
for (int i = 0; i < 1000000; ++i) {
  Foo f;                  // 构造和析构 1000000 次
  f.DoSomthing();
}
```

全局函数和全局变量

- 尽量不用全局函数和全局变量

全局函数和全局变量

- 尽量不用全局函数和全局变量
- 相比单纯为了封装若干个不共享任何静态数据的静态成员函数而创建类，不如使用命名空间

全局函数和全局变量

- 尽量不用全局函数和全局变量
- 相比单纯为了封装若干个不共享任何静态数据的静态成员函数而创建类，不如使用命名空间
- 禁止使用 class 类型的全局变量，允许使用内建类型的全局变量

全局函数和全局变量

- 尽量不用全局函数和全局变量
- 相比单纯为了封装若干个不共享任何静态数据的静态成员函数而创建类，不如使用命名空间
- 禁止使用 class 类型的全局变量，允许使用内建类型的全局变量
- 多线程中的全局变量（静态成员变量）不要使用 class 类型（含 STL 容器）

全局函数和全局变量

- 尽量不用全局函数和全局变量
- 相比单纯为了封装若干个不共享任何静态数据的静态成员函数而创建类，不如使用命名空间
- 禁止使用 class 类型的全局变量，允许使用内建类型的全局变量
- 多线程中的全局变量（静态成员变量）不要使用 class 类型（含 STL 容器）
- 对于全局的字符串常量，使用 C 风格的字符串，不要使用 STL 的字符串

全局函数和全局变量

- 尽量不用全局函数和全局变量
- 相比单纯为了封装若干个不共享任何静态数据的静态成员函数而创建类，不如使用命名空间
- 禁止使用 class 类型的全局变量，允许使用内建类型的全局变量
- 多线程中的全局变量（静态成员变量）不要使用 class 类型（含 STL 容器）
- 对于全局的字符串常量，使用 C 风格的字符串，不要使用 STL 的字符串
- 永远不要使用函数返回值初始化全局变量

将单参数构造函数声明为 explicit

```
Foo::Foo(int num);
void Func(const Foo& test);
Func(2); // oh, oooh, Func(const Foo&) called
```

将单参数构造函数声明为 explicit

```
Foo::Foo(int num);
void Func(const Foo& test);
Func(2); // oh, oooh, Func(const Foo&) called
```

仅在作为数据集合时使用 struct

- 一个只有 set、get 方法的类是不是很烦 ☹

将单参数构造函数声明为 explicit

```
Foo::Foo(int num);
void Func(const Foo& test);
Func(2); // oh, oooh, Func(const Foo&) called
```

仅在作为数据集合时使用 struct

- 一个只有 set、get 方法的类是不是很烦 ☹

为避免拷贝构造函数、赋值操作的滥用和编译器自动生成，可声明其为 private，并无需实现：

```
#define DISALLOW_COPY_AND_ASSIGN(TypeName) \
    TypeName(const TypeName&); \
    void operator=(const TypeName&)
```

构造函数

- 不在构造函数中做太多逻辑相关的初始化
- 构造函数中不易报告错误，不能使用异常
- 操作失败会造成对象初始化失败，引起不确定状态
- 构造函数内调用虚函数，调用不会派发到子类实现中
- 该类型全局变量，构造函数将在 main() 之前被调用，有可能破坏构造函数中暗含的假设条件。例如，google gflags 尚未初始化

默认构造函数

- 新建一个没有参数的对象时，默认构造函数被调用
- 当调用 `new[]` 时，默认构造函数总是被调用
- 编译器提供的默认构造函数不会对变量进行初始化
- 如果定义了其他构造函数，编译器不再提供默认构造函数

真的继承？

- 组合 > 实现继承 > 接口继承 > 私有继承
- 避免使用多重继承，使用时，除一个基类含有实现外，其他基类均为纯接口

真的继承？

- 组合 > 实现继承 > 接口继承 > 私有继承
- 避免使用多重继承，使用时，除一个基类含有实现外，其他基类均为纯接口

声明顺序：public -> protected -> private

1. typedefs 和 enums
2. 常量
3. 构造函数
4. 析构函数
5. 成员函数，含静态成员函数
6. 数据成员，含静态数据成员

函数

- 子类重载的虚函数也要声明 virtual 关键字
- 为降低复杂性，尽量不重载操作符，模板、标准类中使用时提供文档说明
- 存取函数一般内联在头文件中
- 一般不要重载操作符，尤其是赋值操作（`operator=`）
  STL 容器中作为 key，可使用函数类代替 `operator==` 或 `operator<`
- 函数体尽量短小、紧凑，功能单一（不要超过 40 行）

- 不要随意缩写

- 不要随意缩写
- 普通函数、类型（含类与结构体、枚举类型）、常量等使用大小写混合，不含下划线

- 不要随意缩写
- 普通函数、类型（含类与结构体、枚举类型）、常量等使用大小写混合，不含下划线
- 除函数名可适当为动词外，其他命名尽量使用清晰易懂的名词，函数大写开头

- 不要随意缩写
- 普通函数、类型（含类与结构体、枚举类型）、常量等使用大小写混合，不含下划线
- 除函数名可适当为动词外，其他命名尽量使用清晰易懂的名词，函数大写开头
- 宏、枚举等使用全部大写 + 下划线，如 `MY_EXCITING_ENUM_VALUE`。枚举名称属于类型，因此大小写混合：`UrlTableErrors`

- 不要随意缩写
- 普通函数、类型（含类与结构体、枚举类型）、常量等使用大小写混合，不含下划线
- 除函数名可适当为动词外，其他命名尽量使用清晰易懂的名词，函数大写开头
- 宏、枚举等使用全部大写 + 下划线，如 `MY_EXCITING_ENUM_VALUE`。枚举名称属于类型，因此大小写混合：`UrlTableErrors`
- 变量（含类、结极体成员变量）、文件、命名空间、存取函数等使用全部小写 + 下划线

- 不要随意缩写
- 普通函数、类型（含类与结构体、枚举类型）、常量等使用大小写混合，不含下划线
- 除函数名可适当为动词外，其他命名尽量使用清晰易懂的名词，函数大写开头
- 宏、枚举等使用全部大写 + 下划线，如 `MY_EXCITING_ENUM_VALUE`。枚举名称属于类型，因此大小写混合：`UrlTableErrors`
- 变量（含类、结极体成员变量）、文件、命名空间、存取函数等使用全部小写 + 下划线
- 类成员变量以下划线结尾，全局变量以 `g_` 开头，常量以 `k` 开头

- 不要随意缩写
- 普通函数、类型（含类与结构体、枚举类型）、常量等使用大小写混合，不含下划线
- 除函数名可适当为动词外，其他命名尽量使用清晰易懂的名词，函数大写开头
- 宏、枚举等使用全部大写 + 下划线，如 `MY_EXCITING_ENUM_VALUE`。枚举名称属于类型，因此大小写混合：`UrlTableErrors`
- 变量（含类、结极体成员变量）、文件、命名空间、存取函数等使用全部小写 + 下划线
- 类成员变量以下划线结尾，全局变量以 `g_` 开头，常量以 `k` 开头
- 参考现有或相近命名约定（使用业务命名）

```
void ChangeLocalValue();    // ok
void ChgLocVal();           // bad
void ModifyPlayerName();    // ok
void MdfPlyNm();            // toooo bad

class Foo {
 public:
  enum UrlTableErrors {OK = 0, ERROR_OUT_OF_MEMORY};
  static int GetTotalNumber();
  void DoSomething();
  int number() const;
  void set_number(int number);
private:
  int number_;
};
```

- 注释可以炫耀你的成就，也是让你当扛篓子

- 注释可以炫耀你的成就，也是让你当扛篓子
- 注释风格：行注释、块注释均可，但要统一

- 注释可以炫耀你的成就，也是让你当扛篓子
- 注释风格：行注释、块注释均可，但要统一
- 注释要言简意赅，不要拖沓冗余，只要必要的注释

- 注释可以炫耀你的成就，也是让你当扛篓子
- 注释风格：行注释、块注释均可，但要统一
- 注释要言简意赅，不要拖沓冗余，只要必要的注释
- 注释不要太乱，适当缩进

- 注释可以炫耀你的成就，也是让你当扛篓子
- 注释风格：行注释、块注释均可，但要统一
- 注释要言简意赅，不要拖沓冗余，只要必要的注释
- 注释不要太乱，适当缩进
- .h 文件注释功能说明，.cc 文件注释实现说明（如算法等）

- 注释可以炫耀你的成就，也是让你当扛篓子
- 注释风格：行注释、块注释均可，但要统一
- 注释要言简意赅，不要拖沓冗余，只要必要的注释
- 注释不要太乱，适当缩进
- .h 文件注释功能说明，.cc 文件注释实现说明（如算法等）
- TODO：使用全大写的字符串 TODO，后面括号里加上名字、邮件地址

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

函数声明处注释

函数声明处注释

- 函数功能

函数声明处注释

- 函数功能
- 输入及输出

函数声明处注释

- 函数功能
- 输入及输出
- 对类成员函数而言，函数调用期间对象是否需要保持引用参数，是否会释放这些参数

函数声明处注释

- 函数功能
- 输入及输出
- 对类成员函数而言，函数调用期间对象是否需要保持引用参数，是否会释放这些参数
- 如果函数分配了空间，是否需要由调用者释放

函数声明处注释

- 函数功能
- 输入及输出
- 对类成员函数而言，函数调用期间对象是否需要保持引用参数，是否会释放这些参数
- 如果函数分配了空间，是否需要由调用者释放
- 参数是否可以为 NULL

函数声明处注释

- 函数功能
- 输入及输出
- 对类成员函数而言，函数调用期间对象是否需要保持引用参数，是否会释放这些参数
- 如果函数分配了空间，是否需要由调用者释放
- 参数是否可以为 NULL
- 是否存在函数使用的性能隐忧

函数声明处注释

- 函数功能
- 输入及输出
- 对类成员函数而言，函数调用期间对象是否需要保持引用参数，是否会释放这些参数
- 如果函数分配了空间，是否需要由调用者释放
- 参数是否可以为 NULL
- 是否存在函数使用的性能隐忧
- 如果函数是可重入的，其同步前提是什么

自注释的代码

```
bool success = CallFun(interesting_value,
                       10,       // Default base value.
                       false,    // Not first time calling
                       NULL);    // No callback.

const int kDefaultBaseValue = 10;
const bool kFirstTimeCalling = false;
Callback *null_callback = NULL;

bool success = CallFun(interesting_value,
                       kDefaultBaseValue,
                       kFirstTimeCalling,
                       null_callback);
```

- 行宽原则上不超过 80 列，头文件保护可以无视该原则

- 行宽原则上不超过 80 列，头文件保护可以无视该原则
- 包含长路径的可以超出 80 列，尽量避免

- 行宽原则上不超过 80 列，头文件保护可以无视该原则
- 包含长路径的可以超出 80 列，尽量避免
- 一行注释可以包含超过 80 字符的 URL

- 行宽原则上不超过 80 列，头文件保护可以无视该原则
- 包含长路径的可以超出 80 列，尽量避免
- 一行注释可以包含超过 80 字符的 URL
- 如果一个布尔表达式超过标准行宽（80 字符），断行要统一

- 行宽原则上不超过 80 列，头文件保护可以无视该原则
- 包含长路径的可以超出 80 列，尽量避免
- 一行注释可以包含超过 80 字符的 URL
- 如果一个布尔表达式超过标准行宽（80 字符），断行要统一

- 行宽原则上不超过 80 列，头文件保护可以无视该原则
- 包含长路径的可以超出 80 列，尽量避免
- 一行注释可以包含超过 80 字符的 URL
- 如果一个布尔表达式超过标准行宽（80 字符），断行要统一

逻辑不（&&）操作符总位于行尾：

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

- 水平、垂直留白不要乱用

- 水平、垂直留白不要乱用
- 尽量不使用非 ASCII 字符，如果使用的话，参考 UTF-8 格式

- 水平、垂直留白不要乱用
- 尽量不使用非 ASCII 字符，如果使用的话，参考 UTF-8 格式
- 函数参数、逻辑条件、初始化列表：要么所有参数和函数名放在同一行，要么所有参数并排分行；

- 水平、垂直留白不要乱用
- 尽量不使用非 ASCII 字符，如果使用的话，参考 UTF-8 格式
- 函数参数、逻辑条件、初始化列表：要么所有参数和函数名放在同一行，要么所有参数并排分行；
- 除函数定义的左大括号可以置于行首外，包括函数、类、结构体、枚举声明、各种语句的左大括号置于行尾，所有右大括号独立成行

- 水平、垂直留白不要乱用
- 尽量不使用非 ASCII 字符，如果使用的话，参考 UTF-8 格式
- 函数参数、逻辑条件、初始化列表：要么所有参数和函数名放在同一行，要么所有参数并排分行；
- 除函数定义的左大括号可以置于行首外，包括函数、类、结构体、枚举声明、各种语句的左大括号置于行尾，所有右大括号独立成行
- 预处理命令、命名空间不使用额外缩进，类、结构体、枚举、函数、语句使用缩进

- 初始化用 = 还是（）统一就好

- 初始化用 = 还是（）统一就好
- . -> 操作符前后不留空格，* & 不要前后都留

- 初始化用 = 还是 () 统一就好
- . -> 操作符前后不留空格，* & 不要前后都留
- return 不要加 ()

- 初始化用 = 还是 () 统一就好
- . -> 操作符前后不留空格，* & 不要前后都留
- return 不要加 ()
- 如果 default 永不会执行，可以简单的使用assert(false)；空循环体应使用 {} 或 continue，而不是一个简单的分号

- 尽量使用智能指针 `scoped_ptr`
  特点：1) 不能转换所有权 2) 不能共享所有权 3) 不能用于管理数组对象
- 在非常特殊的情况下，如对 STL 容器中对象使用 `shared_ptr`
  特点：1) 引用计数 2) 可以共享对象的所有权 3) STL 可以使用 4) 线程安全
- 任何情况下都不要使用 `auto_ptr`
  特点：1) 两个 `auto_ptr` 不能同时拥有同一个对象 2) 转移赋值

- 禁止使用缺省函数参数
- 引用形参加上 `const`，否则使用指针形参
- 函数重载的使用要清晰、易读，重载函数功能相同
- 禁止使用变长数组（`alloc()`，栈上分配内存，非标准规定）
- 合理使用友元
- 禁止使用异常
- 禁止使用 RTTI(Run-Time Type Informationx)
- 使用 C++ 风格的类型转换，除单元测试外不要使用 `dynamic_cast`

- 能用前置自增/减不用后置自增/减
- `const` 能用则用，提倡 `const` 在前
- 使用确定大小的整型，除位组外不要使用无符号型
- 格式化输出及结构对齐时，注意 32 位和 64 位的系统差异
- C++ 没有规定整型的大小
- 除字符串化、连接外尽量避免使用宏
- 整数用 `0`，实数用 `0.0`，指针用 `NULL`，字符串用 `'\0'`
- 用 `sizeof(varname)` 代替 `sizeof(type)`
- 如果需要一个指针大小的整数，使用 `intptr_t`

```
Foo::Foo(int num = 0) {} // deprecated

++fooIter; // recommend
fooIter++; // deprecated

if (sizeof(short) == 2)                // MAYBE right
if (sizeof(int) == 4)                  // MAYBE right
if (sizeof(long) == 4)                 // MAYBE right
if (sizeof(long long) == 8)            // MAYBE right
if (sizeof(void *) == sizeof(int))     // MAYBE right

double var;
int varSize = sizeof(var);             // recommend
int varSize = sizeof(double);          // deprecated
```

```
int bar = static_cast<int>(var); // recommend
int bar = (int)var;              // deprecated

char* ptr = NULL;
if (a == 0 ||
    abs(var - a) < 0.02 ||
    ptr != NULL || ptr[0] == '\0') {
  ...
}

// endless loops
for (unsigned int i = foo.Length()-1; i >= 0; --i) {
  ...
}
```

关于 Warning

有一个小伙子在一个办公大楼的门口抽着烟，一个妇女路过他身边，并对他说，“你知道不知道这个东西会危害你的健康？我是说，你有没有注意到香烟盒上的那个警告（*Warning*）?”

小伙子说，“没事儿，我是一个程序员”。

那妇女说，“这又怎样?”

程序员说，“我们从来不关心 *Warning*，只关心 *Error*”

```
// which one do you prefer?
if (num == 5)
if (5 == num)
```

```
// which one do you prefer?
if (num == 5)
if (5 == num)
```

编译器将帮你发现问题：

warning: suggest parentheses around assignment used as truth value

```
// which one do you prefer?
if (num == 5)
if (5 == num)
```

编译器将帮你发现问题：

> *warning: suggest parentheses around assignment used as truth value*

建议：计算机能做的交给计算机。不放过任何警告，将编译器（gcc 为例）选项设置为：

```
gcc -Wall -Wextra -Werror
```

如何定义接口：指针还是引用

```
// which one do you prefer?
int func1(const Foo& in, Bar* out);
int func2(const Foo& in, Bar& out);
```

如何定义接口：指针还是引用

```
// which one do you prefer?
int func1(const Foo& in, Bar* out);
int func2(const Foo& in, Bar& out);
```

对于这个方法的使用者来说：

```
Foo foo;
Bar bar;

int ret = func1(foo, &bar);
int ret = func2(foo, bar);
```

如何定义接口：尽量少用 bool

```
// what do false and true mean?
QRegExp rx("moc_*.c", false, true);

// which one do you prefer?
str.replace("%USER%", user, true);
str.replace("%USER%", user, Qt::CaseInsensitive);
```

如何定义接口：尽量少用 bool

```
// what do false and true mean?
QRegExp rx("moc_*.c", false, true);

// which one do you prefer?
str.replace("%USER%", user, true);
str.replace("%USER%", user, Qt::CaseInsensitive);
```

枚举成员至少使用一个枚举的单词

```
enum CaseSensitivity {Insensitive, Sensitive};

// which one do you prefer?
str.replace("%USER%", user, Qt::Insensitive);
str.replace("%USER%", user, Qt::CaseInsensitive);
```

Google Coding Style 特点

- 以人为本
- 规范清晰明了，各种细节都不放过
- 知其然不知其所以然
- 标准规定 VS. 编译器实现
- 正确做法 VS. 习惯做法

Thanks