**google c++编程规范**

**班级：信1604-2 学号：20163786 姓名：宋璋晗**

# **1. 头文件**

通常每一个 .cc 文件都有一个对应的 .h 文件. 也有一些常见例外, 如单元测试代码和只包含 main() 函数的 .cc 文件.

正确使用头文件可令代码在可读性、文件大小和性能上大为改观.

下面的规则将引导你规避使用头文件时的各种陷阱.

## **1.1. Self-contained 头文件**

**Tip**

头文件应该能够自给自足（self-contained,也就是可以作为第一个头文件被引入），以 .h 结尾。至于用来插入文本的文件，说到底它们并不是头文件，所以应以 .inc 结尾。不允许分离出 -inl.h 头文件的做法.

所有头文件要能够自给自足。换言之，用户和重构工具不需要为特别场合而包含额外的头文件。详言之，一个头文件要有 [1.2. #define 保护](https://zh-google-styleguide.readthedocs.io/en/latest/google-cpp-styleguide/headers/" \l "define-guard)，统统包含它所需要的其它头文件，也不要求定义任何特别 symbols.

## **1.2. #define 保护**

**Tip**

所有头文件都应该使用 #define 来防止头文件被多重包含, 命名格式当是: <PROJECT>\_<PATH>\_<FILE>\_H\_ .

为保证唯一性, 头文件的命名应该基于所在项目源代码树的全路径. 例如, 项目 foo 中的头文件 foo/src/bar/baz.h 可按如下方式保护:

#ifndef FOO\_BAR\_BAZ\_H\_#define FOO\_BAR\_BAZ\_H\_...#endif *// FOO\_BAR\_BAZ\_H\_*

## **1.3. 前置声明**

**Tip**

尽可能地避免使用前置声明。使用 #include 包含需要的头文件即可。

****结论：****

* 尽量避免前置声明那些定义在其他项目中的实体.
* 函数：总是使用 #include.
* 类模板：优先使用 #include.

dir/foo.cc 或 dir/foo\_test.cc 的主要作用是实现或测试 dir2/foo2.h 的功能, foo.cc中包含头文件的次序如下:

1. dir2/foo2.h (优先位置, 详情如下)
2. C 系统文件
3. C++ 系统文件
4. 其他库的 .h 文件
5. 本项目内 .h 文件

按字母顺序分别对每种类型的头文件进行二次排序是不错的主意。注意较老的代码可不符合这条规则，要在方便的时候改正它们。

# **2. 作用域**

## **2.1. 匿名命名空间和静态变量**

推荐、鼓励在 .cc 中对于不需要在其他地方引用的标识符使用内部链接性声明，但是不要在 .h 中使用。

匿名命名空间的声明和具名的格式相同，在最后注释上 namespace :

**namespace** {...} *// namespace*

# **3. 类**

类是 C++ 中代码的基本单元. 显然, 它们被广泛使用. 本节列举了在写一个类时的主要注意事项.

## **3.1. 构造函数的职责**

构造函数不允许调用虚函数. 如果代码允许, 直接终止程序是一个合适的处理错误的方式. 否则, 考虑用 Init() 方法或工厂函数.

构造函数不得调用虚函数, 或尝试报告一个非致命错误. 如果对象需要进行有意义的 (non-trivial) 初始化, 考虑使用明确的 Init() 方法或使用工厂模式. Avoid Init() methods on objects with no other states that affect which public methods may be called (此类形式的半构造对象有时无法正确工作).

## **3.2. 结构体 VS. 类**

****总述****

仅当只有数据成员时使用 struct, 其它一概使用 class.

struct 用来定义包含数据的被动式对象, 也可以包含相关的常量, 但除了存取数据成员之外, 没有别的函数功能. 并且存取功能是通过直接访问位域, 而非函数调用. 除了构造函数, 析构函数, Initialize(), Reset(), Validate() 等类似的用于设定数据成员的函数外, 不能提供其它功能的函数.

如果需要更多的函数功能, class 更适合. 如果拿不准, 就用 class.

注意: 类和结构体的成员变量使用不同的 [命名规则](https://zh-google-styleguide.readthedocs.io/en/latest/google-cpp-styleguide/naming/" \l "variable-names).

## **3.3. 继承**

所有继承必须是 public 的. 如果你想使用私有继承, 你应该替换成把基类的实例作为成员对象的方式.

不要过度使用实现继承. 组合常常更合适一些. 尽量做到只在 “是一个” (“is-a”, YuleFox 注: 其他 “has-a” 情况下请使用组合) 的情况下使用继承: 如果 Bar 的确 “是一种” Foo, Bar 才能继承 Foo.

必要的话, 析构函数声明为 virtual. 如果你的类有虚函数, 则析构函数也应该为虚函数.

对于可能被子类访问的成员函数, 不要过度使用 protected 关键字. 注意, 数据成员都必须是 [私有的](https://zh-google-styleguide.readthedocs.io/en/latest/google-cpp-styleguide/classes/" \l "access-control).

## **3.4. 运算符重载**

除少数特定环境外, 不要重载运算符. 也

不要创建用户定义字面量.

只有在意义明显, 不会出现奇怪的行为并且与对应的内建运算符的行为一致时才定义重载运算符. 例如, | 要作为位或或逻辑或来使用, 而不是作为 shell 中的管道.

不要重载 &&, ||, , 或一元运算符 &. 不要重载 operator"", 也就是说, 不要引入用户定义字面量.

## **3.5. 声明顺序**

****总述****

将相似的声明放在一起, 将 public 部分放在最前.

****说明****

类定义一般应以 public: 开始, 后跟 protected:, 最后是 private:. 省略空部分.

在各个部分中, 建议将类似的声明放在一起, 并且建议以如下的顺序: 类型 (包括 typedef, using 和嵌套的结构体与类), 常量, 工厂函数, 构造函数, 赋值运算符, 析构函数, 其它函数, 数据成员.

不要将大段的函数定义内联在类定义中. 通常，只有那些普通的, 或性能关键且短小的函数可以内联在类定义中. 参见 [内联函数](https://zh-google-styleguide.readthedocs.io/en/latest/google-cpp-styleguide/headers/" \l "inline-functions) 一节.

# **4. 函数**

## **4.1. 参数顺序**

****总述****

函数的参数顺序为: 输入参数在先, 后跟输出参数.

****说明****

C/C++ 中的函数参数或者是函数的输入, 或者是函数的输出, 或兼而有之. 输入参数通常是值参或 const 引用, 输出参数或输入/输出参数则一般为非 const 指针. 在排列参数顺序时, 将所有的输入参数置于输出参数之前. 特别要注意, 在加入新参数时不要因为它们是新参数就置于参数列表最后, 而是仍然要按照前述的规则, 即将新的输入参数也置于输出参数之前.

这并非一个硬性规定. 输入/输出参数 (通常是类或结构体) 让这个问题变得复杂. 并且, 有时候为了其他函数保持一致, 你可能不得不有所变通.

## **4.2. 编写简短函数**

我们倾向于编写简短, 凝练的函数.即使一个长函数现在工作的非常好, 一旦有人对其修改, 有可能出现新的问题, 甚至导致难以发现的 bug. 使函数尽量简短, 以便于他人阅读和修改代码.

## **4.3. 引用参数**

****总述****

所有按引用传递的参数必须加上 const.

函数参数列表中, 所有引用参数都必须是 const:

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

总而言之, 大多时候输入形参往往是 const T&. 若用 const T\* 则说明输入另有处理. 所以若要使用 const T\*, 则应给出相应的理由, 否则会使得读者感到迷惑.

## **4.4. 函数重载**

如果打算重载一个函数, 可以试试改在函数名里加上参数信息. 例如, 用 AppendString() 和 AppendInt() 等, 而不是一口气重载多个 Append(). 如果重载函数的目的是为了支持不同数量的同一类型参数, 则优先考虑使用 std::vector 以便使用者可以用 [列表初始化](https://zh-google-styleguide.readthedocs.io/en/latest/google-cpp-styleguide/others/" \l "braced-initializer-list) 指定参数.

## **5.类型转换**

**Tip**

使用 C++ 的类型转换, 如 static\_cast<>(). 不要使用 int y = (int)x 或 int y = int(x) 等转换方式;

不要使用 C 风格类型转换. 而应该使用 C++ 风格.

* 用 static\_cast 替代 C 风格的值转换, 或某个类指针需要明确的向上转换为父类指针时.
* 用 const\_cast 去掉 const 限定符.
* 用 reinterpret\_cast 指针类型和整型或其它指针之间进行不安全的相互转换. 仅在你对所做一切了然于心时使用.

# **6. 命名约定**

最重要的一致性规则是命名管理. 命名的风格能让我们在不需要去查找类型声明的条件下快速地了解某个名字代表的含义: 类型, 变量, 函数, 常量, 宏, 等等, 甚至. 我们大脑中的模式匹配引擎非常依赖这些命名规则.

## **6.1. 通用命名规则**

****总述****

函数命名, 变量命名, 文件命名要有描述性; 少用缩写.

尽可能使用描述性的命名, 别心疼空间, 毕竟相比之下让代码易于新读者理解更重要. 不要用只有项目开发者能理解的缩写, 也不要通过砍掉几个字母来缩写单词.

int price\_count\_reader; *// 无缩写*

int num\_errors; *// "num" 是一个常见的写法*

int num\_dns\_connections; *// 人人都知道 "DNS" 是什么*

int n; *// 毫无意义.*

int nerr; *// 含糊不清的缩写.*

int n\_comp\_conns; *// 含糊不清的缩写.*

int wgc\_connections; *// 只有贵团队知道是什么意思.*

int pc\_reader; *// "pc" 有太多可能的解释了.*

int cstmr\_id; *// 删减了若干字母.*

## **6.2. 文件命名**

****总述****

文件名要全部小写, 可以包含下划线 (\_) 或连字符 (-), 依照项目的约定. 如果没有约定, 那么 “\_” 更好.

可接受的文件命名示例:

* my\_useful\_class.cc
* my-useful-class.cc
* myusefulclass.cc
* myusefulclass\_test.cc // \_unittest 和 \_regtest 已弃用.

## **6.3. 类型命名**

所有类型命名 —— 类, 结构体, 类型定义 (typedef), 枚举, 类型模板参数 —— 均使用相同约定, 即以大写字母开始, 每个单词首字母均大写, 不包含下划线. 例如:

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

*// 类型定义***typedef** hash\_map<UrlTableProperties \*, string> PropertiesMap;

*// using 别名***using** PropertiesMap = hash\_map<UrlTableProperties \*, string>;

*// 枚举***enum** UrlTableErrors { ...

### **普通变量命名**

举例:

string table\_name; *// 好 - 用下划线.*string tablename; *// 好 - 全小写.*

string tableName; *// 差 - 混合大小写*

### **类数据成员**

不管是静态的还是非静态的, 类数据成员都可以和普通变量一样, 但要接下划线.

**class** **TableInfo** {

...

**private**:

string table\_name\_; *// 好 - 后加下划线.*

string tablename\_; *// 好.*

**static** Pool<TableInfo>\* pool\_; *// 好.*};

### **结构体变量**

不管是静态的还是非静态的, 结构体数据成员都可以和普通变量一样, 不用像类那样接下划线:

**struct** UrlTableProperties {

string name;

int num\_entries;

**static** Pool<UrlTableProperties>\* pool;};

## **6.5. 常量命名**

****总述****

声明为 constexpr 或 const 的变量, 或在程序运行期间其值始终保持不变的, 命名时以 “k” 开头, 大小写混合. 例如:

**const** int kDaysInAWeek = 7;

## **6.6. 函数命名**

****总述****

常规函数使用大小写混合, 取值和设值函数则要求与变量名匹配: MyExcitingFunction(), MyExcitingMethod(), my\_exciting\_member\_variable(), set\_my\_exciting\_member\_variable().

一般来说, 函数名的每个单词首字母大写 (即 “驼峰变量名” 或 “帕斯卡变量名”), 没有下划线. 对于首字母缩写的单词, 更倾向于将它们视作一个单词进行首字母大写 (例如, 写作 StartRpc() 而非 StartRPC()).

AddTableEntry()

DeleteUrl()

OpenFileOrDie()

# **7. 注释**

## **7.1. 注释风格**

****总述****

使用 // 或 /\* \*/, 统一就好.

****说明****

// 或 /\* \*/ 都可以; 但 // 更 常用. 要在如何注释及注释风格上确保统一.

## **7.2. 文件注释**

在每一个文件开头加入版权公告.

文件注释描述了该文件的内容. 如果一个文件只声明, 或实现, 或测试了一个对象, 并且这个对象已经在它的声明处进行了详细的注释, 那么就没必要再加上文件注释. 除此之外的其他文件都需要文件注释.

如果一个 .h 文件声明了多个概念, 则文件注释应当对文件的内容做一个大致的说明, 同时说明各概念之间的联系. 一个一到两行的文件注释就足够了, 对于每个概念的详细文档应当放在各个概念中, 而不是文件注释中.

## **7.3. 类注释**

****总述****

每个类的定义都要附带一份注释, 描述类的功能和用法, 除非它的功能相当明显.

*// Iterates over the contents of a GargantuanTable.// Example:// GargantuanTableIterator\* iter = table->NewIterator();// for (iter->Seek("foo"); !iter->done(); iter->Next()) {// process(iter->key(), iter->value());// }// delete iter;***class** **GargantuanTableIterator** {

...};

## **7.4. 函数注释**

****总述****

函数声明处的注释描述函数功能; 定义处的注释描述函数实现.

函数声明处注释的内容:

* 函数的输入输出.
* 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.
* 函数是否分配了必须由调用者释放的空间.
* 参数是否可以为空指针.
* 是否存在函数使用上的性能隐患.
* 如果函数是可重入的, 其同步前提是什么?
* 但也要避免罗罗嗦嗦, 或者对显而易见的内容进行说明. 下面的注释就没有必要加上 “否则返回 false”, 因为已经暗含其中了:

## **7.5. 实现注释**

对于代码中巧妙的, 晦涩的, 有趣的, 重要的地方加以注释.

### **代码前注释**

巧妙或复杂的代码段前要加注释. 比如:

### **行注释**

比较隐晦的地方要在行尾加入注释. 在行尾空两格进行注释. 比如:

## **7.6. 标点, 拼写和语法**

****总述****

注意标点, 拼写和语法; 写的好的注释比差的要易读的多.

****说明****

注释的通常写法是包含正确大小写和结尾句号的完整叙述性语句. 大多数情况下, 完整的句子比句子片段可读性更高. 短一点的注释, 比如代码行尾注释, 可以随意点, 但依然要注意风格的一致性.

虽然被别人指出该用分号时却用了逗号多少有些尴尬, 但清晰易读的代码还是很重要的. 正确的标点, 拼写和语法对此会有很大帮助.

# **格式**

## **8.1. 行长度**

****总述****

每一行代码字符数不超过 80.

我们也认识到这条规则是有争议的, 但很多已有代码都遵照这一规则, 因此我们感觉一致性更重要.

## **8.2. 非 ASCII 字符**

****总述****

尽量不使用非 ASCII 字符, 使用时必须使用 UTF-8 编码.

## **8.3. 函数声明与定义**

****总述****

返回类型和函数名在同一行, 参数也尽量放在同一行, 如果放不下就对形参分行, 分行方式与 [函数调用](https://zh-google-styleguide.readthedocs.io/en/latest/google-cpp-styleguide/formatting/" \l "function-calls) 一致.如果同一行文本太多, 放不下所有参数：

ReturnType LongClassName::ReallyReallyReallyLongFunctionName(

Type par\_name1, *// 4 space indent*

Type par\_name2,

Type par\_name3) {

DoSomething(); *// 2 space indent*

...}

注意以下几点:

* 使用好的参数名.
* 只有在参数未被使用或者其用途非常明显时, 才能省略参数名.
* 如果返回类型和函数名在一行放不下, 分行.
* 如果返回类型与函数声明或定义分行了, 不要缩进.
* 左圆括号总是和函数名在同一行.
* 函数名和左圆括号间永远没有空格.
* 圆括号与参数间没有空格.
* 左大括号总在最后一个参数同一行的末尾处, 不另起新行.
* 右大括号总是单独位于函数最后一行, 或者与左大括号同一行.
* 右圆括号和左大括号间总是有一个空格.
* 所有形参应尽可能对齐.
* 缺省缩进为 2 个空格.
* 换行后的参数保持 4 个空格的缩进.

## **8.4. 函数调用**

函数调用遵循如下形式：

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

如果同一行放不下, 可断为多行, 后面每一行都和第一个实参对齐, 左圆括号后和右圆括号前不要留空格：

bool retval = DoSomething(averyveryveryverylongargument1,

argument2, argument3);

参数也可以放在次行, 缩进四格：

**if** (...) {

...

...

**if** (...) {

DoSomething(

argument1, argument2, *// 4 空格缩进*

argument3, argument4);

}

## **8.5. 条件语句**

倾向于不在圆括号内使用空格. 关键字 if 和 else 另起一行.

**if** (condition) { *// 圆括号里没有空格.*

... *// 2 空格缩进.*

} **else** **if** (...) { *// else 与 if 的右括号同一行.*

...

} **else** {

...

}

通常, 单行语句不需要使用大括号, 如果你喜欢用也没问题; 复杂的条件或循环语句用大括号可读性会更好. 也有一些项目要求 if 必须总是使用大括号:

**if** (condition)

DoSomething(); *// 2 空格缩进.*

**if** (condition) {

DoSomething(); *// 2 空格缩进.*}

## **8.6. 循环和开关选择语句**

switch 语句可以使用大括号分段, 以表明 cases 之间不是连在一起的. 在单语句循环里, 括号可用可不用. 空循环体应使用 {} 或 continue.

**switch** (var) {

**case** 0: { *// 2 空格缩进*

... *// 4 空格缩进*

**break**;

}

**case** 1: {

...

**break**;

}

**default**: {

assert(false);

}

}

## **8.7. 指针和引用表达式**

句点或箭头前后不要有空格. 指针/地址操作符 (\*, &) 之后不能有空格.

下面是指针和引用表达式的正确使用范例:

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

注意:

* 在访问成员时, 句点或箭头前后没有空格.
* 指针操作符 \* 或 & 后没有空格.

## **8.8. 函数返回值**

****总述****

不要在 return 表达式里加上非必须的圆括号.

只有在写 x = expr 要加上括号的时候才在 return expr; 里使用括号.

## **8.9. 变量及数组初始化**

您可以用 =, () 和 {}, 以下的例子都是正确的：

int x = 3;

int x(3);

int x{3};

string name("Some Name");

string name = "Some Name";

string name{"Some Name"};

## **8.10. 类格式**

****总述****

访问控制块的声明依次序是 public:, protected:, private:, 每个都缩进 1 个空格.

****说明****

类声明 (下面的代码中缺少注释, 参考 [类注释](https://zh-google-styleguide.readthedocs.io/en/latest/google-cpp-styleguide/comments/" \l "class-comments)) 的基本格式如下:

**class** **MyClass** : **public** OtherClass {

**public**: *// 注意有一个空格的缩进*

MyClass(); *// 标准的两空格缩进*

**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\_;};