

# 软件开发代码规范

(仅供内部使用)

# 文件修改记录

修改日期	版本	修改页码、章节、条款	修改描述	作者
2024-6-17	v1.0		对外首版	wuhao
2024-6-19	V1.1	2-1 、 4-2 、 6-2 、 6-5 、 7-2 、 7-4	评审修改  1. explicit 单参构造函数 2. define防护符格式去掉项目前缀 3. 推荐项高亮 4. 增加章节9 5. 文件命名，明确后缀 6. 常量定义优先级高于全局和static 修饰的变量 7. 文件注释模版，函数注释模版	wuhao
2024-6-24	V1.2	3-1 、 3-2 、 3-3 、 8-3 、 8-15	1. 修改描述3-1 std改为标准命名空间 2. 修改描述3-2 3. 高亮3-3 4. 8-3推荐工具使用 5. 8-15推荐	

# 目录

目录 .....	3
第一章 原则 .....	5
第二章 头文件 .....	6
2.1 #define 防护符 .....	6
2.2 导入你的依赖 .....	6
2.3 内联函数 .....	7
第三章 作用域 .....	8
3.1 命名空间 .....	8
3.2 内部链接 .....	9
3.3 非成员函数、静态成员函数和全局函数 .....	9
3.4 局部变量 .....	10
第四章 类和函数 .....	11
4.1 构造函数的职责 .....	11
4.2 隐式类型转换 .....	11
4.3 结构体 VS. 类 .....	11
4.4 存取控制 .....	12
4.5 函数的输入输出 .....	12
4.6 缺省参数 .....	12
第五章 其他 C++ 特性 .....	13
5.1 变长数组和 alloca() .....	13
5.2 类型转换 .....	13
5.3 整型 .....	14
5.4 0, nullptr 和 NULL .....	14
5.5 const 用法 .....	14
5.6 auto .....	14
5.7 列表初始化 .....	14
第六章 命名约定 .....	15
6.1 通用命名规则 .....	15
6.2 文件命名 .....	15
6.3 类型命名 .....	15

6.4 变量命名 .....	16
6.5 常量命名 .....	16
6.6 函数命名 .....	16
6.7 命名空间命名 .....	17
6.8 枚举命名 .....	17
6.9 宏命名 .....	17
第七章 注释 .....	18
7.1 注释风格 .....	18
7.2 文件注释 .....	18
7.3 类注释 .....	18
7.4 函数注释 .....	19
7.5 变量注释 .....	19
7.6 实现注释 .....	19
7.7 TODO 注释 .....	20
第八章 格式 .....	21
8.1 行长度 .....	21
8.2 空格还是制表位 .....	21
8.3 函数声明与定义 .....	21
8.4 函数调用 .....	22
8.5 条件语句 .....	22
8.6 循环和开关选择语句 .....	23
8.7 指针和引用表达式 .....	23
8.8 布尔表达式 .....	24
8.9 函数返回值 .....	24
8.10 预处理指令 .....	24
8.11 类格式 .....	25
8.12 构造函数初始值列表 .....	25
8.13 命名空间格式化 .....	26
8.14 水平留白 .....	26
8.15 垂直留白 .....	28
第九章 注意事项 .....	29

## 第一章 原则

规则的作用就是避免混乱。但规则本身一定要权威，有说服力，并且是理性的。我们所见过的大部分编程规范，其内容或不够严谨，或阐述过于简单，或带有一定的武断性。

Google 保持其一贯的严谨精神，我们参考 Google 规范和公司内部工程师的建议，论证严密。形成符合公司实际情况的代码规范，其中部分规范要求开发人员必须遵守，部分规则推荐开发人员尽量遵守。

规则指导书不仅列出你要怎么做，还告诉你为什么要这么做，哪些情况下可以不这么做，以及如何权衡其利弊，团队未必要完全遵照指南，只供参考。

关键在于**保持一致**。

## 第二章 头文件

### 2.1 #define 防护符

- 【规则2-1-1】所有头文件都应该用 #define 防护符来防止重复导入。防护符的格式是：<路径>\_<文件名>\_H\_。

****baz.h**** #ifndef ULSOUND_H_ #define ULSOUND_H_ ... #endif // ULSOUND_H_	****baz.h**** #ifndef SRC_COLLECT_NODE_ULSOUND_INCLUDE_ULSOUND_H_ #define SRC_COLLECT_NODE_ULSOUND_INCLUDE_ULSOUND_H_ ... #endif // SRC_COLLECT_NODE_ULSOUND_INCLUDE_ULSOUND_H_
--	---

### 2.2 导入你的依赖

- 【规则2-2-1】若代码文件或头文件引用了其他地方定义的符号 (symbol)，该文件应该直接导入提供该符号的声明 (declaration) 或者定义 (definition) 的头文件。不应该为了其他原因而导入头文件。

****a.h**** class A; ...  ****b.h**** #include "a.h" class B; ...  ****b.cpp**** #include "b.h"  A a; b b; ...	****a.h**** class A; ...  ****b.h**** #include "a.h" class B; ...  ****b.cpp**** #include "a.h" #include "b.h"  A a; b b; ...
--	--

## 2.3 内联函数

- 【规则2-3-1】只把 10 行以下的小函数定义为内联 (inline)。

\*\*\*\*a.h\*\*\*\*

```
class A {  
public:  
    int getData();  
private:  
    int data_;  
}
```

\*\*\*\*a.cpp\*\*\*\*

```
int A::getData() {  
    return data_;  
}
```

\*\*\*\*a.h\*\*\*\*

```
class A {  
public:  
    inline int getData() {  
        return data_;  
    }  
private:  
    int data_;  
}
```

## 第三章 作用域

### 3.1 命名空间

clang-tidy check 项 : google-build-using-namespace | google-build-namespaces | google-global-names-in-headers

- 【规则 3-1-1】遵守 [命名空间](#) 命名。
- 【规则 3-1-2】用注释给命名空间收尾。

<pre>// .h 文件 namespace mynamespace {  // 所有声明都位于命名空间中 // 注意没有缩进 class MyClass { public: ... void Foo(); };  } 差, 没有注释  // .cc 文件 namespace mynamespace {  // 函数定义位于命名空间中 void MyClass::Foo() { ... }  } 差, 没有注释</pre>	<pre>// .h 文件 namespace mynamespace {  // 所有声明都位于命名空间中 // 注意没有缩进 class MyClass { public: ... void Foo(); };  } // namespace mynamespace  // .cc 文件 namespace mynamespace {  // 函数定义位于命名空间中 void MyClass::Foo() { ... }  } // namespace mynamespace</pre>
--	--

- 【规则 3-1-3】不要在标准的命名空间内声明任何东西，例如 std。

<pre>// .cc 文件 namespace std { ... void getData() {} } // namespace std</pre>	<pre>// .cc 文件 namespace mynamespace { ... void getData() {} } // namespace mynamespace</pre>
---	---

- 【规则 3-1-4】禁止使用 using 指令引入命名空间的所有符号。

<pre>// .cc 文件 using namespace std;  cout &lt;&lt; "test"&lt;&lt;endl;</pre>	<pre>// .cc 文件 namespace mynamespace { ... std::cout&lt;&lt; "test"&lt;&lt;std::endl;  } // namespace mynamespace</pre>
--	---



- 【规则 3-1-5】不要让头文件引入命名空间别名。

<pre>// .h 文件 namespace baz = ::foo::bar::baz;  ...</pre>	<pre>// 在 .cc 中, 用别名缩略常用的名称 namespace baz = ::foo::bar::baz;  // 在 .h 中, 用别名缩略常用的命名空间 namespace librarian {  inline void my_inline_function() {     // 一个函数 (f 或方法) 中的局部别名     namespace baz = ::foo::bar::baz;     ... } } // namespace librarian</pre>
---	--

## 3.2 内部链接

- 【规则 3-2-1】若 .cpp 文件中的定义不会被其他文件使用, 这些定义可以放入匿名命名空间或者声明为 static。

<pre>*****a.cpp*****  void getData() {} // 外部没有使用 ...</pre>	<pre>*****a.cpp*****  namespace { void getData() {} ... } // namespace</pre>
<pre>*****a.cpp*****  void getData() {} // 外部没有使用 ...</pre>	<pre>*****a.cpp*****  static void getData() {} ...</pre>

- 【规则 3-2-2】不要在 .h 文件中使用内部链接。

<pre>*****a.h*****  namespace { void getData() {} ... } // namespace</pre>	<pre>*****a.h*****  namespace my_space { void getData() {} ... } // namespace my_space</pre>
--	--

## 3.3 非成员函数、静态成员函数和全局函数

- 【规则 3-3-1】建议将非成员 (nonmember) 函数放入命名空间。

<pre>*****a.cpp*****  void getData() {} ...</pre>	<pre>*****a.cpp*****  namespace my_space { void getData() {} ... } // namespace my_space</pre>
---	--

- 【规则 3-3-2】尽量不要使用完全全局的函数 (completely global function)。

*****a.cpp*****  void getData() {} ...	*****a.cpp*****  static void getData() {} ...
---	--

### 3.4 局部变量

clang-tidy check 项 : clang-analyzer-core.uninitialized.Assign

- 【规则 3-4-1】应该尽可能缩小函数里的变量的作用域 (scope)，并在声明的同时初始化。

int i; i = f(); // 不好: 初始化和声明分离	int i = f(); // 良好: 声明时初始化
int jobs = NumJobs(); // 更多代码... f(jobs); // 不好: 初始化和使用位置分离	int jobs = NumJobs(); f(jobs); // 良好: 初始化以后立即 (或很快) 使用
vector<int> v; v.push_back(1); // 用花括号初始化更好 v.push_back(2);	vector<int> v = {1, 2}; // 良好: 立即初始化 v
void function(int data) { int count = 10; ... // demo if (true) { std::cout << count + count << std::endl; } }	void function(int data) { ... // demo if (true) { int count = 10; std::cout << count + count << std::endl; } }

## 第四章 类和函数

### 4.1 构造函数的职责

- 【规则4-1-1】不要在构造函数中调用虚函数。

<pre>class A { public:     A();     bool init();     virtual void show(); }  *****a.cpp***** A::A() {     init();     show(); // 不行，调用虚函数 }</pre>	<pre>class A { public:     A();     bool init();     void show(); }  *****a.cpp***** A::A() {     init();     show(); }</pre>
---	---

### 4.2 隐式类型转换

clang-tidy check 项 : google-explicit-constructor

- 【规则4-2-1】不要定义隐式类型转换,对于转换运算符和单参数构造函数,请使用 `explicit` 关键字。

<pre>class Foo {     Foo(int x);     ... };  void Func(Foo f);  Func(42); // 不好,隐式转换,不会报错</pre>	<pre>class Foo {     explicit Foo(int x);     ... };  void Func(Foo f);  Func(42); // Error</pre>
---	---

### 4.3 结构体 VS. 类

- 【规则4-3-1】只有数据成员时使用 `struct`, 其它一概使用 `class`。

<pre>struct Foo {     Foo(int x, double y);     void getData(){}     ... };</pre>	<pre>class Foo {     Foo(int x, double y);     void getData(){}     ... };</pre>
<pre>class Foo {     int width_;     int length_;     ... };</pre>	<pre>struct Foo {     int width;     int length;     ... };</pre>

## 4.4 存取控制

clang-tidy check 项 : `cppcoreguidelines-non-private-member-variables-in-classes`

- 【规则4-4-1】将所有数据成员声明为`private`，除非是`static`成员变量。

<pre>class Foo { public:     Foo(int x, double y); public:     int data_; // 不好，应该定义为私有，提供访问接口     ... };</pre>	<pre>class Foo {     Foo(int x, double y);     int getData() {return data_;} private:     int data_;     ... }; // static 成员变量 class Foo { public:     Foo(int x, double y);     int getData() {return data_;};     static int data_;     ... };</pre>
---	--

## 4.5 函数的输入输出

clang-tidy check 项 : `readability-const-return-type`

- 【规则4-5-1】我们倾向于按值返回，否则按引用返回。避免返回裸指针。

<pre>void getData(int *data) {     *data = 10; } // 不好，倾向于返回值</pre>	<pre>int getData() {     return 10; }</pre>
<pre>const string shorterString(const string s1,const string s2) {     return s1.size()&lt;s2.size()?s1:s2; } // 倾向于返回引用</pre>	<pre>const string &amp;shorterString(const string &amp;s1,const string &amp;s2) {     return s1.size()&lt;s2.size()?s1:s2; }</pre>

- 【规则4-5-2】避免按值返回用`const`修饰。

<pre>const int foo(); // 不好，按值返回加 const</pre>	<pre>int foo();</pre>
---	-----------------------

## 4.6 缺省参数

clang-tidy check 项 : `google-default-arguments`

- 【规则4-6-1】禁止虚函数缺省参数。

<pre>class Base{ public:     virtual void show(int x, int y = 10) {         std::cout &lt;&lt; "base x=" &lt;&lt; x &lt;&lt; ", y=" &lt;&lt; y &lt;&lt; std::endl;     } }; class Child: public Base{ public:     virtual void show(int x, int y = 11) {         std::cout &lt;&lt; "child x=" &lt;&lt; x &lt;&lt; ", y=" &lt;&lt; y &lt;&lt; std::endl;     } // 差，虚函数缺省参数 }; // main Base *pb = new Child(); pb-&gt;show(12); // out child x=12, y=10</pre>	<pre>class Base{ public:     virtual void show(int x, int y) {         std::cout &lt;&lt; "base x=" &lt;&lt; x &lt;&lt; ", y=" &lt;&lt; y &lt;&lt; std::endl;     } }; class Child: public Base{ public:     virtual void show(int x, int y) {         std::cout &lt;&lt; "child x=" &lt;&lt; x &lt;&lt; ", y=" &lt;&lt; y &lt;&lt; std::endl;     } }; // main Base *pb = new Child(); pb-&gt;show(12, 12); // out child x=12, y=12</pre>
---	--

## 第五章 其他 C++ 特性

### 5.1 变长数组和 `alloca()`

- 【规则5-1-1】我们不允许使用变长数组和 `alloca()`。

<code>int* array = alloca(10* sizeof(int));</code>	<code>int array[10]</code>
--	----------------------------

### 5.2 类型转换

`clang-tidy` 选项: `google-readability-casting / cppcoreguidelines-pro-type-cstyle-cast`

- 【规则5-2-1】使用 C++ 的类型转换, `static_cast` `reinterpret_cast` `const_cast` `dynamic_cast`

<pre>float x = 1.2; int y = (int)x</pre>	<pre>float x = 1.2; int y = static_cast&lt;int&gt;(x);</pre>
<pre>int a = 10; const int&amp; b = a; int&amp; c = b; // 错误, 和常量指针不允许给普通 指针赋值或者初始化一样</pre>	<pre>int a = 10; const int&amp; b = a; int&amp; c = const_cast&lt;int&amp;&gt;(b);</pre>
<pre>class Base { public:     Base() { b_val = 1; }     ~Base() {}     virtual void fun() {}     int b_val; };  class Son :public Base { public:     Son() { s_val = 2; }     ~Son() {}     int s_val; };  int main() {     Base* b_ptr = new Base();     Son* s_ptr = (Son*)b_ptr;      return 0; }</pre>	<pre>class Base { public:     Base() { b_val = 1; }     ~Base() {}     virtual void fun() {}     int b_val; };  class Son :public Base { public:     Son() { s_val = 2; }     ~Son() {}     int s_val; };  int main() {     Base* b_ptr = new Base();     Son* s_ptr = dynamic_cast&lt;Son*&gt;(b_ptr);      return 0; }</pre>
<pre>struct S2 {     int a; private:     int b; } s2;  int* p2 = reinterpret_cast&lt;int*&gt;(&amp;s2); // reinterpret_cast 不 更改 p2 的值为 “指向 s2 的指针”</pre>	<pre>struct S1 {     int a; } s1; int* p1 = reinterpret_cast&lt;int*&gt;(&amp;s1);  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, 仅在你对所作一切了然 于心时使用</pre>

## 5.3 整型

- 【规则5-3-1】小心整型类型转换和整型溢出。

<pre>int a = 100; unsigned int b = 2147483647; int c = a + b; // 溢出</pre>	<pre>unsigned int a = 100; unsigned int b = 2147483647; unsigned int c = a + b; // 不会溢出</pre>
---	---

- 【规则5-3-2】<stdint.h> 定义了 int16\_t, uint32\_t, int64\_t 等整型，在需要确保整型大小时可以使用它们代替 short, unsigned long long 等。在 C 整型中，只使用 int。在合适的情况下，推荐使用标准类型如 size\_t 和 ptrdiff\_t。

<pre>short a = 0; // 不好 unsigned int b = 0; // 不好 unsigned int b = 2147483647; int c = a + b; // 溢出</pre>	<pre>int16_t a = 0; // 好 uint32_t b = 0; uint32_t b = 2147483647; uint32_t c = a + b; // 不会溢出</pre>
<pre>vector&lt;int&gt; v; int size = v.size();</pre>	<pre>vector&lt;int&gt; v; size_t size = v.size();</pre>
<pre>int *a=new int(1); int *b=new int(2); int *result=a-b;</pre>	<pre>int *a=new int(1); int *b=new int(2); ptrdiff_t result=a-b;</pre>

## 5.4 0, nullptr 和 NULL

clang-tidy check 项: modernize-use-nullptr

- 【规则5-4-1】指针使用 nullptr，字符使用 '\0'（而不是 0 字面值）。

<pre>int *pName = NULL;</pre>	<pre>int *pName = nullptr;</pre>
<pre>char ch = 0;</pre>	<pre>char ch = '\0'</pre>

## 5.5 const 用法

clang-tidy check 项: performance-unnecessary-value-param

- 【规则5-5-1】我们强烈建议你在任何可能的情况下都要使用 const。

<pre>void printData(std::string str) {     std::cout &lt;&lt; "str="&lt;&lt;str&lt;&lt;std::endl; } // 没有改变 str,推荐加 const 修饰</pre>	<pre>void printData(const std::string&amp; str) {     std::cout &lt;&lt; "str="&lt;&lt;str&lt;&lt;std::endl; } // 没有改变 str,推荐加 const 修饰</pre>
--	---

## 5.6 auto

clang-tidy check 项: modernize-use-auto

- 【规则5-6-1】不影响代码可读性的情况下，推荐使用 auto。

<pre>sparse_hash_map&lt;string, int&gt; m; sparse_hash_map&lt;string, int&gt;::iterator iter = m.find(val);</pre>	<pre>sparse_hash_map&lt;string, int&gt; m; auto iter = m.find(val);</pre>
---	---

## 5.7 列表初始化

- 【规则5-7-1】千万别直接列表初始化 auto 变量。

<pre>auto d = {1.23}; // d 即是 std::initializer_list&lt;double&gt;,让人误解</pre>	<pre>auto d = double{1.23}; // d 即为 double, 并非 std::initializer_list</pre>
--	--

## 第六章 命名约定

### 6.1 通用命名规则

- 【规则6-1-1】函数命名，变量命名，文件命名要有描述性；少用缩写。

<pre>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; // 不知道是什么意思 int pc_reader;   // "pc" 有太多可能的解释了 int cstmr_id;    // 删减了若干字母</pre>	<pre>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</pre>
--	---

### 6.2 文件命名

- 【规则6-2-1】源文件名要全部小写，只能是小写加下划线(\_)，头文件后缀(.h)，源文件后缀(.cpp)。

<pre>myErrFileName.cpp my-useful-class.cc myusefulclass.cc</pre>	<pre>my_useful_class.cpp</pre>
--	--------------------------------

- 【规则6-2-2】不要使用已经存在于 /usr/include 下的文件名。

<pre>string.h // 自定义头文件和系统重复</pre>	<pre>my_string.h</pre>
------------------------------------	------------------------

### 6.3 类型命名

- 【规则6-3-1】类型名称的每个单词首字母均大写，不包含下划线。

<pre>class myClass { ... class my_class { ...</pre>	<pre>// 类和结构体 class UrlTable { ... class UrlTableTester { ... struct UrlTableProperties { ...  // 类型定义 typedef hash_map&lt;UrlTableProperties *, string&gt; PropertiesMap;  // using 别名 using PropertiesMap = hash_map&lt;UrlTableProperties *, string&gt;;  // 枚举 enum UrlTableErrors { ...</pre>
---	--

## 6.4 变量命名

- 【规则6-4-1】变量（包括函数参数）和数据成员名一律小写，单词之间用下划线连接。类的成员变量以下划线结尾，但结构体的就不用加下划线。

<pre>string tableName; // 差 - 混合大小写</pre>	<pre>普通变量命名 string table_name; // 好 - 用下划线. string tablename; // 好 - 全小写.  类数据成员 class TableInfo { ... private: string table_name_; // 好 - 后加下划线. string tablename_; // 好. static Pool&lt;TableInfo&gt;* pool_; // 好. };  结构体变量 struct UrlTableProperties { string name; int num_entries; static Pool&lt;UrlTableProperties&gt;* pool; };</pre>
---	---

- 【规则6-4-2】静态变量增加 s\_ 作为前缀，全局变量增加 g\_ 作为前缀。

<pre>static int count = 0;</pre>	<pre>static int s_count = 0;</pre>
<pre>int count = 0;</pre>	<pre>int g_count = 0;</pre>
<pre>static int count = 0; // 不好，全局静态变量</pre>	<pre>static int sg_count = 0;</pre>

## 6.5 常量命名

- 【规则6-5-1】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。如果同时包含static const修饰，优先级（常量 > static和全局变量）

<pre>const int daysInAWeek = 7;</pre>	<pre>const int kDaysInAWeek = 7;</pre>
<pre>const static int s_days_in_a_week = 7; // 不好，常量优先级高</pre>	<pre>const static int kDaysInAWeek = 7;</pre>

## 6.6 函数命名

clang-tidy check 项: google-objc-function-naming

- 【规则6-6-1】常规函数使用大小写混合。

<pre>getTime() // 不好，需要大小写混合</pre>	<pre>GetTime() // 好</pre>
<pre>class Object { public: void GetMember(); // 不好, 要求与变量名匹配 private: int exciting_member_variable_; };</pre>	<pre>class Object { public: void GetExcitingMemberVariable private: int exciting_member_variable_; };</pre>



- 【规则6-6-2】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。

```
StartRPC()
```

```
StartRpc()
```

## 6.7 命名空间命名

- 【规则6-7-1】推荐命名空间以小写字母命名，最高级命名空间的名字取决于项目名称。

```
namespace imu {
...
}
```

```
namespace x9_collect_node_imu {
...
} // x9_collect_node_imu
```

- 【规则6-7-2】要避免嵌套的命名空间与常见的顶级命名空间发生名称冲突。

```
namespace x9_collect_node_imu {

namespace std { // 差，与标准空间冲突

...
} // std
} // x9_collect_node_imu
```

```
namespace x9_collect_node_imu {

namespace imu_index {

...
} // imu_index
} // x9_collect_node_imu
```

## 6.8 枚举命名

- 【规则6-8-1】枚举的命名应当和常量或宏一致，单独的枚举值应该优先采用常量的命名方式。

```
enum UriTableErrors {
    ok = 0,
    error,
    failed,
};
```

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

## 6.9 宏命名

- 【规则6-9-1】全部大写，使用下划线，必须使用括号扩起表达式。

```
#define Round(x) ...
#define PI_rounded 3.0 // 没有全部大写
#define MAX(a, b) a > b ? a : b // 差，没有括号
```

```
#define ROUND(x) ...
#define PI_ROUNDED (3.0)
#define MAX(a, b) ((a) > (b) ? (a) : (b))
```

## 第七章 注释

### 7.1 注释风格

- 【规则 7-1-1】注释风格，// 或 /\* \*/ 都可以；但 // 更常用。

```
*****a.cpp*****  
int a = 0; // 变量 a  
int b = 0; /* 变量 b*/
```

```
*****a.cpp*****  
int a = 0; // 变量 a  
int b = 0; // 变量 b
```

### 7.2 文件注释

- 【规则 7-2-1】推荐在每一个文件开头加入版权公告，法律公告和作者信息等文件内容。（推荐vsode 插件：Doxygen Documentation Generator。使用方法：文件第一行输入 `/** + 回车`，自动生成如下文件注释）

```
*****a.h*****  
// my file
```

```
*****a.h*****  
  
/**  
 * @file record_msg.h  
 * @author your name (you@domain.com)  
 * @brief 记录传感器数据  
 * @version 0.1  
 * @date 2024-06-19  
 *  
 * @copyright Copyright 2023 HJ Technology Co.Ltd.  
 All rights reserved  
 *  
 */
```

### 7.3 类注释

- 【规则 7-3-1】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。

```
*****a.h*****  
class GargantuanTableIterator {  
 ...  
};
```

```
*****a.h*****  
  
// Iterates over the contents of a GargantuanTable.  
class GargantuanTableIterator {  
 ...  
};
```

## 7.4 函数注释

- 【规则 7-4-1】基本上每个函数声明处前都应当加上注释，描述函数的功能和用途。只有在函数的功能简单而明显时才能省略这些注释。（推荐 vsode 插件：Doxygen Documentation Generator。使用方法：函数名上一行输入 `/**` + `回车`，自动生成如下函数注释）

<pre>*****a.h***** int MyType::getData(int value) {     std::cout &lt;&lt; "append " &lt;&lt; value &lt;&lt; std::endl;     return value * 2; }</pre>	<pre>*****a.h*****  /**  * @brief Get the Data object  *  * @param value xxxx  * @return int xxxxx  */ int MyType::getData(int value) {     std::cout &lt;&lt; "append " &lt;&lt; value &lt;&lt; std::endl;     return value * 2; }</pre>
---	---

- 【规则 7-4-2】一眼就能看出函数的功能，不需要额外注释。

<pre>// Returns true if the table cannot hold any more entries.(不需要注释) bool IsTableFull();</pre>	<pre>*****a.h*****  bool IsTableFull();</pre>
--	---

## 7.5 变量注释

- 【规则 7-5-1】通常变量名本身足以很好说明变量用途。某些情况下，也需要额外的注释说明。

<pre>int a; // 不能表意，也没有注释</pre>	<pre>int num_total_entries_; // 记录进入总数</pre>
---------------------------------	--

## 7.6 实现注释

- 【规则 7-6-1】代码前注释，推荐复杂代码添加注释。

	<pre>// Divide result by two, taking into account that x // contains the carry from the add. for (int i = 0; i &lt; result-&gt;size(); i++) {     x = (x &lt;&lt; 8) + (*result)[i];     (*result)[i] = x &gt;&gt; 1;     x &amp;= 1; }</pre>
--	---

## 7.7 TODO 注释

clang-tidy check 项: `google-readability-todo`

- **【规则 7-7-1】** TODO 注释要使用全大写的字符串 TODO，在随后的圆括号里写上你的名字，邮件地址，bug ID，或其它身份标识和与这一 TODO 相关的 issue，圆括号内至少有你的名字。

// TODO: Use a "\*" here for concatenation operator.

// TODO(kl@gmail.com): Use a "\*" here for concatenation operator.

// TODO(Zeke) change this to use relations.

// TODO(bug 12345): remove the "Last visitors" feature

## 第八章 格式

### 8.1 行长度

- 【规则 8-1-1】每一行代码字符数不超过 80，或者不超过 120。

<pre>printf("test num=%d, num2=%d, num3=%d, num4=%d, num5=%d\n", num, num2, num3, num4, num5);</pre>
<pre>printf("test num=%d, num2=%d, num3=%d, num4=%d, num5=%d\n",       num, num2, num3, num4, num5);</pre>

### 8.2 空格还是制表位

- 【规则 8-2-1】只使用空格，每次缩进 2 个空格。

### 8.3 函数声明与定义

- 【规则 8-3-1】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一致。推荐 clang-format 进行一键化处理

详细说明：

1. 使用表意明确的参数名。
2. 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。
3. 左大括号总在最后一个参数同一行的末尾处，不另起新行。
4. 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。
5. 右圆括号和左大括号间总是有一个空格。
6. 所有形参应尽可能对齐。
7. 默认缩进为 2 个空格。
8. 换行后的参数保持 4 个空格的缩进。

<pre>int SetCount(int a); // 不好, a 表意不明确</pre>	<pre>int SetCount(int count)</pre>
<pre>int SetCount ( int count ) {     count_ = count;} // 不好, 左大括号没有和函数名同行, 函数名与左圆括号有空格, 参数与圆括号之间有空格, 右大括号没有单独一行</pre>	<pre>int SetCount(int count) {     count_ = count; }</pre>
<pre>int ClassName::FunctionName(Type par_name1,                              Type par_name2); // 不好</pre>	<pre>int ClassName::FunctionName(Type par_name1,                              Type par_name2) { // 4 个空格缩进     DoSomething();     ... }</pre>

## 8.4 函数调用

- 【规则 8-4-1】要么一行写完函数调用，要么在圆括号里对参数分行，要么参数另起一行且缩进四格。如果没有其它顾虑的话，尽可能精简行数，比如把多个参数适当地放在同一行里。

<pre>int SetParam(int count, int time, int length, int width); // 不好，一行可以写完不要分行，分行，没有缩进 4 个空格</pre>
<pre>int SetParam(int count, int time, int length, int width);</pre>

- 【规则 8-4-2】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。参数也可以放在次行，缩进四格。

<pre>DoSomething( const std::string&amp; argument1, const std::string&amp; argument2, const std::string&amp;argument3, argument4); // 一行放不下</pre>
<pre>DoSomething(     argument1, argument2, // 4 空格缩进     argument3, argument4);  DoSomething(argument1, argument2, //与第一个实参对齐             argument3, argument4);</pre>

- 【规则 8-4-3】如果一些参数本身就是略复杂的表达式，且降低了可读性，那么可以直接创建临时变量描述该表达式，并传递给函数。

<pre>bool retval = DoSomething((scores[x] * y + bases[x]), x, y, z);</pre>	<pre>int my_heuristic = scores[x] * y + bases[x]; bool retval = DoSomething(my_heuristic, x, y, z);</pre>
--	---

- 【规则 8-4-4】如果一系列参数本身就有一定的结构，可以酌情地按其结构来决定参数格式。

<pre>// 通过 3x3 矩阵转换 widget. my_widget.Transform(x1, x2, x3, y1, y2, y3,                     z1, z2, z3);</pre>	<pre>// 通过 3x3 矩阵转换 widget. my_widget.Transform(x1, x2, x3,                     y1, y2, y3,                     z1, z2, z3);</pre>
--	--

## 8.5 条件语句

clang-tidy check 项: `google-readability-braces-around-statements`

- 【规则 8-5-1】倾向于不在圆括号内使用空格，关键字 `if` 和 `else` 另起一行。

<pre>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.     ... }</pre>	<pre>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 {     ... }</pre>
---	--

- 【规则 8-5-2】所有情况下 `if`, `else`, `else if` 和左圆括号间都有个空格。右圆括号和左大括号之间也要有个空格。

```
if(condition) // 差 - IF 后面没空格.
if (condition){ // 差 - { 前面没空格.
if(condition){ // 变本加厉地差.
```

```
if (condition) { // 好 - IF 和 { 都与空格紧邻
} else if (continue1)
```

- 【规则 8-5-3】`if` 必须总是使用大括号。

```
if (condition)
DoSomething(); // 没有大括号.
```

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

## 8.6 循环和开关选择语句

- 【规则 8-6-1】如果有不满足 `case` 条件的枚举值, `switch` 应该总是包含一个 `default` 匹配。如果 `default` 应该永远执行不到, 简单的加条 `assert`。

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

```
switch (var) {
case 0: { // 2 空格缩进
... // 4 空格缩进
break;
}
default: {
assert(false);
}
}
```

- 【规则 8-6-2】在单语句循环里, 必须总是使用大括号。

```
for (int i = 0; i < kSomeNumber; ++i)
printf("I love you\n");
```

```
for (int i = 0; i < kSomeNumber; ++i) {
printf("I take it back\n");
}
```

- 【规则 8-6-3】空循环体应使用 `{}` 或 `continue`, 而不是一个简单的分号。

```
while (condition); // 差
```

```
while (condition) {
// 反复循环直到条件失效.
}
for (int i = 0; i < kSomeNumber; ++i) {} // 可 - 空循环体.
while (condition) {
continue;
} // 可 - continue 表明没有逻辑.
```

## 8.7 指针和引用表达式

- 【规则 8-7-1】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。

```
x = * p;
p = & x;
```

```
x = *p;
p = &x;
```



- 【规则 8-7-2】在声明指针变量或参数时，星号与类型或变量名紧挨都可以，在单个文件内要保持风格一致。

int x, *y; // 不允许在多重声明中不能使用 & 或 * char *c; // 差 - * 两边都有空格 const string &str; // 差 - & 两边都有空格.	// 好, 空格前置. char *c; const string &str;  // 好, 空格后置. char* c; const string& str;
--	--

## 8.8 布尔表达式

- 【规则 8-8-1】如果一个布尔表达式超过标准行宽，断行方式要统一一下，比如逻辑与（&&）操作符均位于行尾或者均位于行头。

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

## 8.9 函数返回值

- 【规则 8-9-1】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。

return (value); return(result);	return result; // 可以用圆括号把复杂表达式圈起来，改善可读性。 return (some_long_condition && another_condition);
------------------------------------	--

## 8.10 预处理指令

- 【规则 8-10-1】预处理指令不要缩进，从行首开始。

// 差 - 指令缩进 if (lopsided_score) { #if DISASTER_PENDING // 差 - "#if" 应该放在行 开头 DropEverything(); #endif // 差 - "#endif" 不要缩进 BackToNormal(); }	// 好 - 指令从行首开始 if (lopsided_score) { #if DISASTER_PENDING // 正确 - 从行首开始 DropEverything(); # if NOTIFY // 非必要 - # 后跟空格 NotifyClient(); # endif #endif BackToNormal(); }
---	---



## 8.11 类格式

- 【规则 8-11-1】访问控制块的声明依次序是 `public:`, `protected:`, `private:`, 每个都缩进 1 个空格。

```
class MyClass : public OtherClass {
private:
    bool SomeInternalFunction();

    int some_var_;
    int some_other_var_;

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

}; // 声明顺序不满足 public:, protected:,
private
```

```
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_;
};
```

## 8.12 构造函数初始值列表

- 【规则 8-12-1】构造函数初始化列表放在同一行或按四格缩进并排多行。

```
// 不能放在同一行,没有缩进 4 个空格
MyClass::MyClass(int var)
: some_var_(var), some_other_var_(var + 1) {
    DoSomething();
}

// 如果初始化列表需要置于多行,将每一个
// 成员放在单独的一行,没有逐行对齐
MyClass::MyClass(int var)
: some_var_(var),          // 4 space indent
  some_other_var_(var + 1) { // lined up
    DoSomething();
}
```

```
// 如果所有变量能放在同一行:
MyClass::MyClass(int var) : some_var_(var) {
}

// 如果不能放在同一行,必须置于冒号后,并
// 缩进 4 个空格
MyClass::MyClass(int var)
: some_var_(var), some_other_var_(var + 1) {
    DoSomething();
}

// 如果初始化列表需要置于多行,将每一个成
// 员放在单独的一行,并逐行对齐
MyClass::MyClass(int var)
: some_var_(var),          // 4 space indent
  some_other_var_(var + 1) { // lined up
    DoSomething();
}

// 右大括号 } 可以和左大括号 { 放在同一行
// 如果这样做合适的话
MyClass::MyClass(int var)
: some_var_(var) {}
```

## 8.13 命名空间格式化

- 【规则 8-13-1】命名空间内容不缩进。

<pre>namespace {      // 错, 缩进多余了     void foo() {         ...     }  } // namespace</pre>	<pre>namespace {      void foo() { // 正确. 命名空间内没有额外的缩进         ...     }  } // namespace</pre>
--	--

- 【规则 8-13-2】声明嵌套命名空间时, 每个命名空间都独立成行。

<pre>namespace foo {     namespace bar {         ...     } // namespace bar } // namespace foo</pre>	<pre>namespace foo { namespace bar {     ... } // namespace bar } // namespace foo</pre>
--	--

## 8.14 水平留白

- 【规则 8-14-1】通用规则, 水平留白的使用根据在代码中的位置决定, 永远不要在行尾添加没意义的留白。

<pre>void f(bool b){ // 左大括号没有空格. ... int i = 0 ; // 分号前加空格.  // 继承与初始化列表中的冒号前后没有空格. class Foo:public Bar { public:     // 对于单行函数的实现, 在大括号内加上空格, 然后是函数实现     void Reset() {baz_ = 0;} // 没有空格把大括号与实现分开.</pre>	<pre>void f(bool b) { // 左大括号前总是有空格. ... int i = 0; // 分号前不加空格. // 列表初始化中大括号内的空格是可选的. // 如果加了空格, 那么两边都要加上. int x[] = { 0 }; int x[] = {0};  // 继承与初始化列表中的冒号前后恒有空格. class Foo : public Bar { public:     // 对于单行函数的实现, 在大括号内加上空格     // 然后是函数实现     Foo(int b) : Bar(), baz_(b) {} // 大括号里面是空的话, 不加空格.     void Reset() { baz_ = 0; } // 用空格把大括号与实现分开.</pre>
---	---

● 【规则 8-14-2】循环和条件语句。

<pre> if (b){      // 圆括号和大括号之间没有空格 }else{      // else 前后没有空格. } while ( test ){ // 差, 圆括号内部有额外空格. for ( int i = 0; i &lt; 5; ++i ){ switch ( i ){ // 循环和条件语句的圆括号里有空格 for ( int i = 0; i &lt; 5; ++i ){ for ( ; i &lt; 5; ++i ){ // 循环里内; 后没有空格 switch ( i ){ case 1 :      // switch case 的冒号前有空格. ... case 2:break; // 如果冒号有代码, 没加空格. </pre>	<pre> if (b) {      // if 条件语句和循环语句关键字后均有空格. } else {      // else 前后有空格. } while (test) {} // 圆括号内部不加空格. switch (i) { for (int i = 0; i &lt; 5; ++i) { switch (i) { // 循环和条件语句的圆括号里不加空格 for ( int i = 0; i &lt; 5; ++i ){ for ( ; i &lt; 5; ++i ){ // 循环里内; 后恒有空格; 前可以加个空格. switch (i) { case 1:      // switch case 的冒号前无空格. ... case 2: break; // 如果冒号有代码, 加个空格. </pre>
---	--

● 【规则 8-14-3】操作符。

<pre> // 赋值运算符前后没有空格. x=0;  // 在参数和一元操作符之间加空格. x = - 5; ++ x; if (x &amp;&amp; ! y) ... </pre>	<pre> // 赋值运算符前后总是有空格. x = 0;  // 其它二元操作符也前后恒有空格, 不过对于表达式的子式可以不加空格. // 圆括号内部没有紧邻空格. v = w * x + y / z; v = w*x + y/z; v = w * (x + z);  // 在参数和一元操作符之间不加空格. x = -5; ++x; if (x &amp;&amp; !y) ... </pre>
--	--

● 【规则 8-14-4】模版和转换。

<pre> // 尖括弧内有空格. vector&lt; string &gt; x; y = static_cast&lt; char* &gt;(x);  // 在类型与指针操作符之间留空格也可以, 但要保持一致, 例子*位置不一致. vector&lt;char *&gt; x; vector&lt;char* &gt; y; </pre>	<pre> // 尖括弧(&lt; and &gt;) 不与空格紧邻, &lt; 前没有空格, &gt; 和 ( 之间也没有. vector&lt;string&gt; x; y = static_cast&lt;char*&gt;(x);  // 在类型与指针操作符之间留空格也可以, 但要保持一致. vector&lt;char *&gt; x; vector&lt;char*&gt; y; </pre>
--	---

## 8.15 垂直留白

- 【规则 8-15-1】推荐不在万不得已，不要使用空行。尤其是：两个函数定义之间的空行不要超过 2 行，函数体首尾不要留空行，函数体中也不要随意添加空行。

```
// 函数之间空行太多

int xxxxx::start() {
// 函数首不要空行

    int ret = RET_OK;

    if (i2c_.initialize() < 0) {
        HJ_ERROR("can not open file %s", DEV_PATH);
        init_status_ = false;
        ret = RET_ERR;
    } else {
        loop_timer_ = hj_bf::HJCreateTimer("pressure_timer",
frequency_ * 1000, &PressureSensorWF::pressure_timer,
this);
    }

    return ret;

// 函数尾不要空行
}
```

```
int xxxxx::start() {
    int ret = RET_OK;
    if (i2c_.initialize() < 0) {
        HJ_ERROR("can not open file %s", DEV_PATH);
        init_status_ = false;
        ret = RET_ERR;
    } else {
        loop_timer_ = hj_bf::HJCreateTimer("pressure_timer",
frequency_ * 1000, &PressureSensorWF::pressure_timer,
this);
    }
    return ret;
}
```

## 第九章 注意事项

1. 文档高亮部分为推荐您这样做，目前不强制要求
2. vsode 安装 cpplint 插件有助于暴露出不符合规范的代码，但是默认配置未经筛选，不一定完全符合本次代码指导规则。