|  |  |  |  |
| --- | --- | --- | --- |
|  | **文 档 编 号** | **文档版本** | **密级** |
| **SW\_2\_4** | **1.0** | **密** |
| **开发适用** | | **共 20 页** |

**软件开发代码规范**

**(仅供内部使用)**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 拟制： |  |  | 日期： |  |
| 审核： |  |  | 日期： |  |
| 签发： |  |  | 日期： |  |
|  | | | | |

# 目录

[目录 2](#_Toc365074239)

[第一章 原则 5](#_Toc2057955300)

[第二章 头文件 6](#_Toc482881521)

[2.1 自给自足的头文件 6](#_Toc2001717947)

[2.2 #define 防护符 6](#_Toc1573197186)

[2.3 导入你的依赖 6](#_Toc929703453)

[2.4 内联函数 7](#_Toc982991998)

[第三章 作用域 8](#_Toc2108993833)

[3.1 命名空间 8](#_Toc89831240)

[3.2 内部链接 9](#_Toc1384945194)

[3.3 非成员函数、静态成员函数和全局函数 9](#_Toc1483561618)

[3.4 局部变量 10](#_Toc1216144580)

[第四章 类 11](#_Toc73955407)

[6.1 构造函数的职责 11](#_Toc1626929790)

[6.2 隐式类型转换 11](#_Toc433288254)

[6.3 结构体 VS. 类 11](#_Toc1924857955)

[第五章 其他 C++ 特性 12](#_Toc952477055)

[5.1 变长数组和 alloca() 12](#_Toc30394876)

[5.2 类型转换 12](#_Toc2104015164)

[5.3 整型 13](#_Toc896398780)

[5.4 0, nullptr 和 NULL 13](#_Toc782684718)

[5.5 列表初始化 13](#_Toc1254838714)

[第六章 命名约定 15](#_Toc1122963266)

[6.1 常量命名 15](#_Toc616685847)

[6.2 文件命名 15](#_Toc1597472065)

[6.3 类型命名 15](#_Toc1104306597)

[6.4 变量命名 16](#_Toc2068971958)

[6.5 常量命名 16](#_Toc1897380819)

[6.6 函数命名 16](#_Toc1480894893)

[6.7 命名空间命名 17](#_Toc1873949773)

[6.8 枚举命名 17](#_Toc608435033)

[6.9 宏命名 17](#_Toc1845969132)

[第七章 注释 18](#_Toc1784421425)

[7.1 注释风格 18](#_Toc1091316554)

[7.2 文件注释 18](#_Toc1700203432)

[7.3 类注释 19](#_Toc1210134963)

[7.4 函数注释 19](#_Toc2021020007)

[7.5 变量注释 19](#_Toc535711782)

[7.6 实现注释 20](#_Toc1171645148)

[7.7 TODO 注释 20](#_Toc2110851247)

[第八章 格式 21](#_Toc1920656976)

[8.1 行长度 21](#_Toc507723118)

[8.2 非 ASCII 字符 21](#_Toc1179512180)

[8.3 空格还是制表位 21](#_Toc1994612384)

[8.4 函数声明与定义 21](#_Toc2134652908)

[8.5 函数调用 22](#_Toc1612800434)

[8.6 条件语句 22](#_Toc1771986691)

[8.7 循环和开关选择语句 23](#_Toc939646316)

[8.8 指针和引用表达式 24](#_Toc1643195310)

[8.9 布尔表达式 24](#_Toc1728518207)

[8.10 函数返回值 25](#_Toc1836045096)

[8.11 预处理指令 25](#_Toc278396380)

[8.12 类格式 25](#_Toc835873274)

[8.13 构造函数初始值列表 26](#_Toc811524714)

[8.14 命名空间格式化 26](#_Toc895082227)

[8.15 水平留白 27](#_Toc285861691)

[8.16 垂直留白 28](#_Toc1915831311)

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 |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

# 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, 5 万汉字的指南涉及广泛, 论证严密. 我们翻译该系列指南的主因也正是其严谨性. 严谨意味着指南的价值不仅仅局限于它罗列出的规范, 更具参考意义的是它为了列出规范而做的谨慎权衡过程.

指南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. 其他团队未必要完全遵照指南亦步亦趋, 如前面所说, 这份指南是 Google 根据自身实际情况打造的, 适用于其主导的开源项目. 其他团队可以参照该指南, 或从中汲取灵感, 建立适合自身实际情况的规范.

关键在于**保持一致。**

# 第二章 头文件

## #define 防护符

* 【规则2-2-1】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: <项目>\_<路径>\_<文件名>\_H\_。

|  |  |
| --- | --- |
| \*\*\*\*\*baz.h\*\*\*\*\*  ... | \*\*\*\*\*baz.h\*\*\*\*\*  #ifndef FOO\_BAR\_BAZ\_H\_  #define FOO\_BAR\_BAZ\_H\_  ...  #endif // FOO\_BAR\_BAZ\_H\_ |
| \*\*\*\*\*baz.h\*\*\*\*\*  #ifndef BAZ\_H\_  #define BAZ\_H\_  ...  #endif //BAZ\_H\_ | \*\*\*\*\*baz.h\*\*\*\*\*  #ifndef FOO\_BAR\_BAZ\_H\_  #define FOO\_BAR\_BAZ\_H\_  ...  #endif // FOO\_BAR\_BAZ\_H\_ |

## 导入你的依赖

* 【规则2-3-1】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 (include) 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;  ... |

## 内联函数

* 【规则2-4-1】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

## 命名空间

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

|  |
| --- |
| // .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 |

* **【规则3-1-3】**在导入语句、 gflags 声明/定义以及其他命名空间的类的前向声明 (forward declaration) 之后, 用命名空间包裹整个源代码文件。

|  |
| --- |
| #include "a.h"  DEFINE\_FLAG(bool, someflag, false, "某个旗标");  namespace mynamespace {  using ::foo::Bar;  ...命名空间内的代码... // 代码紧贴左边框.  } // namespace mynamespace |

* **【规则3-1-4】**若要将自动生成的 proto 消息代码放入命名空间, 可以在 .proto 文件中使用 package 修饰符 (specifier)。
* **【规则3-1-5】**不要在 std 命名空间内声明任何东西. 不要前向声明标准库的类。
* **【规则3-1-6】**禁止使用 using 指令 引入命名空间的所有符号。

|  |
| --- |
| // 禁止: 这会污染命名空间.  using namespace foo;  Xxx.xx（）  Foo::xxx.xx() |

* **【规则3-1-7】**除了在明显标注为内部使用的命名空间内, 不要让头文件引入命名空间别名。

|  |
| --- |
| // 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 {  namespace impl { // 仅限内部使用, 不是 API.  namespace sidetable = ::pipeline\_diagnostics::sidetable;  } // namespace impl  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 |

* **【规则3-1-8】**如果命名空间的名称包含 “internal”, 代表用户不应该使用这些 API。
* **【规则3-1-9】**禁止内联命名空间

|  |
| --- |
| // Absl 以外的代码不应该使用这一内部符号.  using ::absl::container\_internal::ImplementationDetail; |

## 内部链接

* **【规则3-2-1】**若其他文件不需要使用 .cc 文件中的定义, 这些定义可以放入匿名命名空间 (unnamed namespace) 或声明为 static, 以实现内部链接 (internal linkage). 但是不要在 .h 文件中使用这些手段。

|  |
| --- |
| namespace {  ...  } // namespace  Int a = 0;  Namespace {  Int a = 0;  }  Static a = 0; |
|  |

## 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。

|  |  |
| --- | --- |
| \*\*\*\*\*a.cpp\*\*\*\*\*  void getData() {}  ... | \*\*\*\*\*a.cpp\*\*\*\*\*  namespace my\_space {  void getData() {}  ...  } // namespace my\_space |

* **【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() {}  ... |

* **【规则3-3-3】**不要仅仅为了给静态成员 (static member) 分组而使用类 (class)。
* **【规则3-3-4】**类的静态方法应当和类的实例或静态数据紧密相关。

## 局部变量

* **【规则3-4-1】**应该尽可能缩小函数变量的作用域 (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. |
| // 低效的实现:  for (int i = 0; i < 1000000; ++i) {  Foo f; // 调用 1000000 次构造函数和析构函数.  f.DoSomething(i);  } | Foo f; // 调用 1 次构造函数和析构函数.  for (int i = 0; i < 1000000; ++i) {  f.DoSomething(i);  } |

# 第四章 类

## 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, 也不要在无法报出错误时进行可能失败的初始化

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

## 隐式类型转换

* 【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。

|  |  |
| --- | --- |
| class Foo {  Foo(int x, double y);  ...  };  void Func(Foo f);  Func({42, 3.14}); //不好，隐式转换 | class Foo {  explicit Foo(int x, double y);  ...  };  void Func(Foo f);  Func({42, 3.14}); // Error |

## 结构体 VS. 类

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

# 第五章 其他 C++ 特性

## 变长数组和 alloca()

* 【规则5-1-1】我们不允许使用变长数组和 alloca()。
* Void func(int in){
  + Char xxx[in];
* }

## 类型转换

* 【规则5-2-1】使用 C++ 的类型转换,static\_cast reinterpret\_cast const\_cast dynamic\_cast

|  |  |
| --- | --- |
| float x = 1.2;  int y = (int)x | float x = 1.2;  int y = static\_cast<int>(x); |
|  | int a = 10;  const int& b = a;  //b = 20;//错误原因：常量引用，不允许修改值  //int& 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  int& c = const\_cast<int&>(b); |
|  | 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<Son\*>(b\_ptr);  return 0;  } |
|  | 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 |

## 整型

* 【规则5-3-1】小心整型类型转换和整型提升

|  |  |
| --- | --- |
| int a = 100;  unsigned int b = 2147483647;  int c = a + b; // 溢出 | unsigned int a = 100;  unsigned int b = 2147483647;  unsigned int c = a + b; // 不会溢出 |

* 【规则5-3-2】如果您的代码涉及容器返回的大小（size），确保其类型足以应付容器各种可能的用法。拿不准时，类型越大越好
* 【规则5-3-3】不要使用 uint32\_t 等无符号整型, 除非你是在表示一个位组而不是一个数值, 或是你需要定义二进制补码溢出. 尤其是不要为了指出数值永不会为负, 而使用无符号类型. 相反, 你应该使用断言来保护数据

|  |  |
| --- | --- |
| for (unsigned int i = foo.Length()-1; i >= 0; --i) {  ...demo // 永远不会退出  } | for (int i = foo.Length()-1; i >= 0; --i) {  assert( foo.Length()-1 > 0);  ...demo  } |

* 【规则5-3-4】<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
* 【规则5-3-5】如果已知整数不会太大, 我们常常会使用 int, 如循环计数. 在类似的情况下使用原生类型 int. 你可以认为 int 至少为 32 位, 但不要认为它会多于 32 位. 如果需要 64 位整型, 用 int64\_t 或 uint64\_t.

## 0, nullptr 和 NULL

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

|  |  |
| --- | --- |
| int \*pName = NULL; | int \*pName = nullptr; |
| char ch = 0; | char ch = ‘\0’ |

## 列表初始化

* 【规则5-5-1】C++11 中，任何对象类型都可以被列表初始化。

|  |
| --- |
| // Vector 接收了一个初始化列表。  vector<string> v{"foo", "bar"};  // 不考虑细节上的微妙差别，大致上相同。  // 您可以任选其一。  vector<string> v = {"foo", "bar"};  // 可以配合 new 一起用。  auto p = new vector<string>{"foo", "bar"};  // map 接收了一些 pair, 列表初始化大显神威。  map<int, string> m = {{1, "one"}, {2, "2"}};  // 初始化列表也可以用在返回类型上的隐式转换。  vector<int> test\_function() { return {1, 2, 3}; }  // 初始化列表可迭代。  for (int i : {-1, -2, -3}) {}  // 在函数调用里用列表初始化。  void TestFunction2(vector<int> v) {}  TestFunction2({1, 2, 3}); |

* 【规则5-5-2】用户自定义类型也可以定义接收 std::initializer\_list<T> 的构造函数和赋值运算符，以自动列表初始化。

|  |
| --- |
| class MyType {  public:  // std::initializer\_list 专门接收 init 列表。  // 得以值传递。  MyType(std::initializer\_list<int> init\_list) {  for (int i : init\_list) append(i);  }  MyType& operator=(std::initializer\_list<int> init\_list) {  clear();  for (int i : init\_list) append(i);  }  };  MyType m{2, 3, 5, 7}; |

* 【规则5-5-3】用列表初始化也适用于常规数据类型的构造，哪怕没有接收 std::initializer\_list<T> 的构造函数。

|  |
| --- |
| double d{1.23};  // MyOtherType 没有 std::initializer\_list 构造函数，  // 直接上接收常规类型的构造函数。  class MyOtherType {  public:  explicit MyOtherType(string);  MyOtherType(int, string);  };  MyOtherType m = {1, "b"};  // 不过如果构造函数是显式的（explict），您就不能用 `= {}` 了。  MyOtherType m{"b"}; |

* 【规则5-5-4】千万别直接列表初始化 auto 变量

|  |  |
| --- | --- |
| auto d = {1.23}; | auto d = double{1.23}; |

# 命名约定

## 常量命名

* 【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。

|  |  |
| --- | --- |
| 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. | 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 |

## 文件命名

* 【规则6-2-1】函文件名要全部小写, 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。
* 【规则6-2-2】C++ 文件要以 .cc or .cpp 结尾, 头文件以 .h 结尾.
* 【规则6-2-3】不要使用已经存在于 /usr/include 下的文件名

|  |  |
| --- | --- |
| myErrFileName.cpp | my\_useful\_class.cc  my-useful-class.cc  myusefulclass.cc |

## 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

|  |  |
| --- | --- |
| class myClass { ...  class my\_class { ... | // 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 { ... |

## 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

|  |  |
| --- | --- |
| string tableName; // 差 - 混合大小写 | 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<TableInfo>\* pool\_; // 好.  };  结构体变量  struct UrlTableProperties {  string name;  int num\_entries;  static Pool<UrlTableProperties>\* pool;  }; |

* 【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

|  |  |
| --- | --- |
| static int count = 0; | static int s\_count = 0; |
| int count = 0; | int g\_count = 0; |

## 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

|  |  |
| --- | --- |
| const int daysInAWeek = 7; | const int kDaysInAWeek = 7; |

## 函数命名

* 【规则6-6-1】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(驼峰变量名)
* my\_exciting\_member\_variable\_
* set\_my\_exciting\_member\_variable()
* 【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() | AddTableEntry()  DeleteUrl()  OpenFileOrDie()  StartRpc() |

## 命名空间命名

* 【规则6-7-1】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. 要注意避免嵌套命名空间的名字之间和常见的顶级命名空间的名字之间发生冲突

|  |  |
| --- | --- |
| namespace imu {  ...  } | namespace 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 (添加例子)
* 【规则6-7-3】（去掉）对于 internal 命名空间, 要当心加入到同一 internal 命名空间的代码之间发生冲突

|  |  |
| --- | --- |
|  |  |

## 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 UrlTableErrors {  ok = 0,  error,  failed,  }; | enum UrlTableErrors {  kOK = 0,  kErrorOutOfMemory,  kErrorMalformedInput,  };  enum AlternateUrlTableErrors {  OK = 0,  OUT\_OF\_MEMORY = 1,  MALFORMED\_INPUT = 2,  }; |

## 宏命名

* 【规则6-9-1】（宏中内容用括号括起来）全部大写, 使用下划线

|  |  |
| --- | --- |
| #define Round(x) ...  #define PI\_rounded 3.0 | #define ROUND(x) ...  #define PI\_ROUNDED (3.0) |

# 注释

## 注释风格

* **【规则7-1-1】**注释风格, // 或 /\* \*/ 都可以; 但 // 更 常用. ~~要在如何注释及注释风格上确保统一。~~

|  |  |
| --- | --- |
| \*\*\*\*\*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 |

## 文件注释

* **【规则7-2-1】推介**在每一个文件开头加入版权公告, 法律公告和作者信息,文件内容

|  |  |
| --- | --- |
| \*\*\*\*\*a.h\*\*\*\*\*  // my file | \*\*\*\*\*a.h\*\*\*\*\*  / /Components for manipulating sequences of characters -\*- C++ -\*-  // Copyright (C) 1997-2019 Free Software Foundation, Inc.  //  // This file is part of the GNU ISO C++ Library. This library is free  // software; you can redistribute it and/or modify it under the  // terms of the GNU General Public License as published by the  // Free Software Foundation; either version 3, or (at your option)  // any later version.  // This library is distributed in the hope that it will be useful,  // but WITHOUT ANY WARRANTY; without even the implied warranty of  // MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the  // GNU General Public License for more details.  // Under Section 7 of GPL version 3, you are granted additional  // permissions described in the GCC Runtime Library Exception, version  // 3.1, as published by the Free Software Foundation.  // You should have received a copy of the GNU General Public License and  // a copy of the GCC Runtime Library Exception along with this program;  // see the files COPYING3 and COPYING.RUNTIME respectively. If not, see  // <http://www.gnu.org/licenses/>. |

## 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。（注释我们给简单点）

|  |  |
| --- | --- |
| \*\*\*\*\*a.h\*\*\*\*\*  class GargantuanTableIterator {  ...  }; | \*\*\*\*\*a.h\*\*\*\*\*  // 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 {  ...  }; |

## 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。（查看**规则7-4-2**）
* **【规则7-4-2】**避免罗罗嗦嗦, 或者对显而易见的内容进行说明。

|  |  |
| --- | --- |
| \*\*\*\*\*a.h\*\*\*\*\*  int MyType::getData(int value) {  std::cout << "append " << value << std::endl;  return value \* 2;  }  // Returns true if the table cannot hold any more entries.(罗罗嗦嗦)  bool IsTableFull(); | \*\*\*\*\*a.h\*\*\*\*\*  /\*\*  \* @brief Get the Data object  \*  \* @param value xxxx  \* @return int xxxxx  \*/  int MyType::getData(int value) {  std::cout << "append " << value << std::endl;  return value \* 2;  } |

## 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.

|  |  |
| --- | --- |
|  | int num\_total\_entries\_; // 记录进入总数 |

## 实现注释

* **【规则7-6-1】**代码前注释.（推介复杂的代码带上注释）

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

* **【规则7-6-2】**行注释.（去掉）

|  |  |
| --- | --- |
|  | // 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. |

* **【规则7-6-3】**函数参数注释.（去掉）

|  |  |
| --- | --- |
| // What are these arguments?  const DecimalNumber product = CalculateProduct(values, 7, false, nullptr); | ProductOptions options;  options.set\_precision\_decimals(7);  options.set\_use\_cache(ProductOptions::kDontUseCache);  const DecimalNumber product =  CalculateProduct(values, options, /\*completion\_callback=\*/nullptr); |

## TODO 注释

* **【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.（至少名字）

|  |  |
| --- | --- |
|  | // 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 |

# 格式

## 行长度

* **【规则8-1-1】**每一行代码字符数不超过 80.（推介每一行代码字符数不超过 80，或者120）

|  |
| --- |
| printf(“test num=%d, num2=%d, num3=%d, num4=%d, num5=%d\n”, num, num2, num3, num4, num5); |
| printf(“test num=%d, num2=%d, num3=%d,num4=%d, num5=%d\n”,  num, num2, num3, num4, num5); |

* **【规则8-1-2】**包含长路径的 #include 语句可以超出80列.（去掉）

## 非 ASCII 字符

* **【规则8-2-1】**尽量不使用非 ASCII 字符, 使用时必须使用 UTF-8 编码.（看能不能说清楚）

|  |  |
| --- | --- |
| "\xEF\xBB\xBF" | u8"\uFEFF" |

## 空格还是制表位

* **【规则8-3-1】**只使用空格, 每次缩进 2 个空格.

## 函数声明与定义

* **【规则8-4-1】**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.（提炼下，都是推介）

详细说明：

1. 使用好的参数名.
2. 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.
3. 如果返回类型和函数名在一行放不下, 分行.
4. 如果返回类型与函数声明或定义分行了, 不要缩进.
5. 左圆括号总是和函数名在同一行.
6. 函数名和左圆括号间永远没有空格.
7. 圆括号与参数间没有空格.
8. 左大括号总在最后一个参数同一行的末尾处, 不另起新行.
9. 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.
10. 右圆括号和左大括号间总是有一个空格.
11. 所有形参应尽可能对齐.
12. 缺省缩进为 2 个空格.
13. 换行后的参数保持 4 个空格的缩进.
14. 未被使用的参数如果其用途不明显的话, 在函数定义处将参数名注释起来
15. 未被使用的参数, 或者根据上下文很容易看出其用途的参数, 可以省略参数名

|  |  |
| --- | --- |
| ReturnType ClassName::FunctionName( Type par\_name1, Type par\_name2)  {  DoSomething();  ...  } | ReturnType ClassName::FunctionName(Type par\_name1, Type par\_name2) {  DoSomething();  ...  } |
| void Circle::Rotate(double) {} | void Circle::Rotate(double /\*radians\*/) {} |

## 函数调用

* **【规则8-5-1】**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（反面例子）

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

* **【规则8-5-2】**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（反面例子）

|  |
| --- |
| bool retval = DoSomething(argument1, argument2, argument3); |
| if (...) {  DoSomething(  argument1, argument2, // 4 空格缩进  argument3, argument4);  } |

* **【规则8-5-3】**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（反面例子）

|  |
| --- |
| int my\_heuristic = scores[x] \* y + bases[x];  bool retval = DoSomething(my\_heuristic, x, y, z); |

* **【规则8-5-4】**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（确认格式）（反面例子）

|  |
| --- |
| // 通过 3x3 矩阵转换 widget.  my\_widget.Transform(x1, x2, x3,  y1, y2, y3,  z1, z2, z3); |

## 条件语句

* **【规则8-6-1】**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.  ...  } | 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 {  ...  } |

* **【规则8-6-2】**所有情况下 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 |

* **【规则8-6-3】**如果能增强可读性, 简短的条件语句允许写在同一行. 只有当语句简单并且没有使用 else 子句时使用（删除）

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

* **【规则8-6-4】**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（删除）（加一条强调所有条件 while等都要加大括号括起来）

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

* **【规则8-6-5】**但如果语句中某个 if-else 分支使用了大括号的话, 其它分支也必须使用（删除）

|  |  |
| --- | --- |
| if (condition) {  foo;  } else  bar; | if (condition)  foo;  else {  bar;  } |

## 循环和开关选择语句

* **【规则8-7-1】**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（删除）
* **【规则8-7-2】**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 (如果有输入值没有 case 去处理, 编译器将给出 warning). 如果 default 应该永远执行不到, 简单的加条 assert（反面例子）

|  |
| --- |
| switch (var) {  case 0: { // 2 空格缩进  ... // 4 空格缩进  break;  }  case 1: {  ...  break;  }  default: {  assert(false);  }  } |

* **【规则8-7-3】**在单语句循环里, 括号必须用（反面例子）

|  |
| --- |
| 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8-7-4】**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 - contunue 表明没有逻辑. |

## 指针和引用表达式

* **【规则8-8-1】**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（反面例子）

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

* **【规则8-8-2】**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; |

## 布尔表达式

* **【规则8-9-1】**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) {  ...  } |

## 函数返回值

* **【规则8-10-1】**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); |

## 预处理指令

* **【规则8-11-1】**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();  } |

## 类格式

* **【规则8-12-1】**访问控制块的声明依次序是 public:, protected:, private:, 每个都缩进 1 个空格.（反面例子）

|  |
| --- |
| 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\_;  }; |

## 构造函数初始值列表

* **【规则8-13-1】**构造函数初始化列表放在同一行或按四格缩进并排多行.（反面例子）

|  |
| --- |
| // 如果所有变量能放在同一行:  MyClass::MyClass(int var) : some\_var\_(var) {  DoSomething();  }  // 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) {} |

## 命名空间格式化

* **【规则8-14-1】**命名空间内容不缩进.

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

* **【规则8-14-2】**声明嵌套命名空间时, 每个命名空间都独立成行.

|  |  |
| --- | --- |
| namespace foo {  namespace bar {  ...  } // namespacve bar  } // namespace foo | namespace foo {  namespace bar {  ...  } // namespacve bar  } // namespace foo |

## 水平留白（反面例子）

* **【规则8-15-1】**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.
* **【规则8-15-2】**通用规则.

|  |
| --- |
| 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. |

* **【规则8-15-3】循环和条件语句**.

|  |
| --- |
| 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) {} // 圆括号内部不紧邻空格.  switch (i) {  for (int i = 0; i < 5; ++i) {  switch ( i ) { // 循环和条件语句的圆括号里可以与空格紧邻.  if ( test ) { // 圆括号, 但这很少见. 总之要一致.  for ( int i = 0; i < 5; ++i ) {  for ( ; i < 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. |

* **【规则8-15-4】操作符**.

|  |
| --- |
| // 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 && !y)  ... |

* **【规则8-15-5】模版和转换**.

|  |
| --- |
| // 尖括号(< and >) 不与空格紧邻, < 前没有空格, > 和 ( 之间也没有.  vector<string> x;  y = static\_cast<char\*>(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<char \*> x; |

## 垂直留白

* **【规则8-15-5】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**.

|  |  |
| --- | --- |
| int getData() {  }  // 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;  } |