# C++ 语言编码规范V1.0

## 规则等级划分

| 规则 | 等级 |
| --- | --- |
| 1. 头文件 | |
| 1.1. Self-contained 头文件 | 必须 |
| 1.2. 头文件保护 | 必须 |
| 1.3. 内联函数 | 建议 |
| 1.4. #include 的路径及顺序 | 建议 |
| 2. 作用域 | |
| 2.1. 命名空间 | 必须 |
| 2.2. 匿名命名空间和静态变量 | 建议 |
| 2.3. 非成员函数、静态成员函数和全局函数 | 必须 |
| 2.4. 局部变量 | 必须 |
| 2.5. 静态和全局变量 | 必须 |
| 3. 类 | |
| 3.1. 构造函数的职责 | 建议 |
| 3.2. 隐式类型转换 | 必须 |
| 3.3. 可拷贝类型和可移动类型 | 建议 |
| 3.4. 结构体 VS. 类 | 建议 |
| 3.5. 继承 | 建议 |
| 3.6. 多重继承 | 建议 |
| 3.7. 运算符重载 | 建议 |
| 3.8. 存取控制 | 建议 |
| 3.9. 声明顺序 | |
| 4. 函数 | |
| 4.1. 参数顺序 | 建议 |
| 4.2. 编写简短函数 | 必须 |
| 4.3. 引用参数 | 建议 |
| 4.4. 缺省参数 | 必须 |
| 5. 其他 C++ 特性 | |
| 5.1. 右值引用 | 建议 |
| 5.2. 变长数组和 alloca() | 建议 |
| 5.3. 异常 | 建议 |

| 规则 | 等级 |
| --- | --- |
| 5.4. 运行时类型识别 | 建议 |
| 5.5. 类型转换 | 必须 |
| 5.6. 前置自增和自减 | 建议 |
| 5.7. `const` 用法 | 建议 |
| 5.8. `constexpr` 用法 | 建议 |
| 5.9. 整型 | |
| 5.10. 64 位下的可移植性 | 建议 |
| 5.11. 预处理宏 | 建议 |
| 5.12. 0，nullptr | 建议 |
| 5.13. sizeof | 必须 |
| 5.14. auto | 建议 |
| 5.15. Lambda 表达式 | 建议 |
| 6. 命名约定 | |
| 6.1. 通用命名规则 | 必须 |
| 6.2. 文件命名 | 建议 |
| 6.3. 类型命名 | 必须 |
| 6.4. 变量命名 | 必须 |
| 6.5. 常量命名 | 必须 |
| 6.6. 函数命名 | 必须 |
| 6.7. 命名空间命名 | 必须 |
| 6.8. 枚举命名 | 必须 |
| 6.9. 宏命名 | 必须 |
| 6.10. 目录命名 | 建议 |
| 7. 注释 | 建议 |
| 8. 格式 | |
| 8.1.基本要求 | 必须 |
| 8.2. Lambda 表达式 | 建议 |
| 8.3. 函数调用 | 建议 |
| 8.4. 条件语句 | 必须 |
| 8.5. 循环和开关选择语句 | 必须 |

| 规则 | 等级 |
| --- | --- |
| 8.6. 指针和引用表达式 | 必须 |
| 8.7. 布尔表达式 | 必须 |
| 8.8. 函数返回值 | 建议 |
| 8.9. 预处理指令 | 必须 |
| 8.10. 类格式 | 必须 |
| 8.11. 构造函数初始值列表 | 建议 |
| 8.12. 命名空间格式化 | 建议 |
| 8.13. 垂直留白 | 建议 |
| 8.14. 函数长度 | 建议 |
| 8.15. 文件长度 | 建议 |
| 8.16. 函数代码块缩进 | 建议 |

# 1. 头文件

每一个 `.cc` 文件都要有一个对应的 `.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。

## 1.1. Self-contained 头文件

所有头文件要能够自给自足。
头文件不要定义任何特别 symbols。

## 1.2. 头文件保护

所有头文件都应该使用 `#define` 来防止头文件被多重包含，命名格式是：
`<PROJECT>_<PATH>_<FILE>_H_`。
此外，Linux 可使用#pragma once，Windows可以使用 `#ONCE`。

## 1.3. 内联函数

只有当函数只有 10 行甚至更少时才将其定义为内联函数。

## 1.4. #include 的路径及顺序

`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

举例来说，`google-awesome-project/src/foo/internal/fooserver.cc` 的包含次序如下：

```
#include "foo/public/fooserver.h" // 优先位置

#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"
```

⬆ 返回规则等级划分

# 2. 作用域

## 2.1. 命名空间

鼓励在 `.cc` 文件内使用匿名命名空间或 `static` 声明。使用具名的命名空间时，其名称可基于项目名或相对路径。在头文件中禁止使用 using 指示（using-directive）。禁止使用内联命名空间（inline namespace）。

- 不要在命名空间 `std` 内声明任何东西（hash除外），包括标准库的类前置声明。
- 不应该使用 *using 指示* 引入整个命名空间的标识符号。

```
// 禁止 -- 污染命名空间
using namespace foo;
```

- 不要在头文件中使用 *命名空间别名* 除非显式标记内部命名空间使用。

```
// 在 .cc 中使用别名缩短常用的命名空间
namespace baz = ::foo::bar::baz;
```

```
// 在 .h 中使用别名缩短常用的命名空间
namespace librarian {
namespace impl {  // 仅限内部使用
namespace sidetable = ::pipeline_diagnostics::sidetable;
}  // namespace impl

inline void my_inline_function() {
  // 限制在一个函数中的命名空间别名
  namespace baz = ::foo::bar::baz;
  ...
}
}  // namespace librarian
```

- 禁止用内联命名空间

## 2.2. 匿名命名空间和静态变量

在 `.cc` 文件中定义一个不需要被外部引用的变量时，可以将它们放在匿名命名空间或声明为 `static`。但是不要在 `.h` 文件中这么做。

## 2.3. 非成员函数、静态成员函数和全局函数

使用静态成员函数或命名空间内的非成员函数，不要用裸的全局函数。
举例而言，对于头文件 `myproject/foo_bar.h`，应当使用

```
namespace myproject {
namespace foo_bar {
void Function1();
void Function2();
}  // namespace foo_bar
}  // namespace myproject
```

而非

```
namespace myproject {
class FooBar {
 public:
  static void Function1();
  static void Function2();
};
}  // namespace myproject
```

## 2.4. 局部变量

将函数变量尽可能置于最小作用域内，并在变量声明时进行初始化.比如:

```
int i;
i = f(); // 坏--初始化和声明分离
```

```
int j = g(); // 好--初始化时声明
```

```
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}; // 好--v 一开始就初始化
```

属于 `if`，`while` 和 `for` 语句的变量应当在这些语句中正常地声明，举例而言：

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

## 2.5. 静态和全局变量

禁止定义静态储存周期非POD变量，禁止使用含有副作用的函数初始化POD全局变量，因为多编译单元中的静态变量执行时的构造和析构顺序是未明确的，这将导致代码的不可移植。

禁止使用类的 *静态储存周期* 变量：由于构造和析构函数调用顺序的不确定性，它们会导致难以发现的 bug。不过 `constexpr` 变量除外，毕竟它们又不涉及动态初始化或析构。

静态生存周期的对象，即包括了全局变量，静态变量，静态类成员变量和函数静态变量，都必须是原生数据类型 (POD : Plain Old Data)： 即 int，char 和 float，以及 POD 类型的指针、数组和结构体。

静态变量的构造函数、析构函数和初始化的顺序在 C++ 中是只有部分明确的，甚至随着构建变化而变化，导致难以发现的 bug。 所以除了禁用类类型的全局变量，我们也不允许用函数返回值来初始化 POD 变量，除非该函数（比如 `getenv()` 或 `getpid()` ）不涉及任何全局变量。函数作用域里的静态变量除外，毕竟它的初始化顺序是有明确定义的，而且只会在指令执行到它的声明那里才会发生。

**↑ 返回规则等级划分**

# 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

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。

## 3.2. 隐式类型转换

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。

## 3.3. 可拷贝类型和可移动类型

如果你的类型需要，就让它们支持拷贝 / 移动。否则，就把隐式产生的拷贝和移动函数禁用。

## 3.4. 结构体 VS. 类

仅当只有数据成员时使用 `struct`，其它一概使用 `class`。

## 3.5. 继承

使用组合 (YuleFox 注: 这一点也是 GoF 在 `<<Design Patterns>>` 里反复强调的) 常常比使用继承更合理。如果使用继承的话，定义为 `public` 继承。

## 3.6. 多重继承

真正需要用到多重实现继承的情况少之又少。只在以下情况我们才允许多重继承：最多只有一个基类是非抽象类；其它基类都是以 `Interface` 为后缀的 *纯接口类* 。

## 3.7. 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。

不要为了避免重载操作符而走极端。比如说，应当定义 `==` ，`=` ，和 `<<` 而不是 `Equals()` ，`CopyFrom()` 和 `PrintTo()` 。 反过来说，不要只是为了满足函数库需要而去定义运算符重载。比如说，如果你的类型没有自然顺序，而你要将它们存入 `std::set` 中，最好还是定义一个自定义的比较运算符而不是重载 `<` 。

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。

类型转换运算符在 3.2. 隐式类型转换 一节有提及。 `=` 运算符在 3.3. 可拷贝类型和可移动类型 一节有提及。

## 3.8. 存取控制

将 *所有* 数据成员声明为 `private` ，除非是 `static const` 类型成员 (遵循 6.5. 常量命名 规则)。处于技术上的原因，在使用 Google Test 时我们允许测试固件类中的数据成员为 protected。

### 3.9. 声明顺序

将相似的声明放在一起，将 `public` 部分放在最前。

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 1.3. 内联函数 一节。

**↑ 返回规则等级划分**

## 4. 函数

### 4.1. 参数顺序

函数的参数顺序为：输入参数在先，后跟输出参数。

### 4.2. 编写简短函数

我们倾向于编写简短，凝练的函数。建议控制在40行以内，小函数化，提高可阅读性、易维护性。

### 4.3. 引用参数

所有按引用传递的参数必须加上 `const` 。

### 4.4. 缺省参数

只允许在非虚函数中使用缺省参数，且必须保证缺省参数的值始终一致。 缺省参数与函数重载遵循同样的规则。一般情况下建议使用函数重载，尤其是在缺省函数带来的可读性提升不能弥补下文中所提到的缺点的情况下。

**↑ 返回规则等级划分**

## 5. 其他 C++ 特性

### 5.1. 右值引用

只在定义移动构造函数与移动赋值操作时使用右值引用。

### 5.2. 变长数组和 alloca()

不允许使用变长数组和 `alloca()` 。

建议改用更安全的分配器（allocator），就像 `std::vector` 或 `std::unique_ptr<T[]>` 。

### 5.3. 异常

异常只能抛出 `std::exception` 及其子类

### 5.4. 运行时类型识别

避免使用 RTTI。

RTTI 有合理的用途但是容易被滥用，因此在使用时请务必注意。在单元测试中可以使用 RTTI，但是在其他代码中请尽量避免。尤其是在新代码中，使用 RTTI 前务必三思。

如果程序能够保证给定的基类实例实际上都是某个派生类的实例，那么就可以自由使用 dynamic_cast. 在这种情况下，使用 dynamic_cast 也是一种替代方案。

不要去手工实现一个类似 RTTI 的方案。反对 RTTI 的理由同样适用于这些方案，比如带类型标签的类继承体系。而且，这些方案会掩盖你的真实意图。

## 5.5. 类型转换

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。

## 5.6. 前置自增和自减

对于迭代器和其他模板对象使用前缀形式 (`++i`) 的自增，自减运算符。

## 5.7. `const` 用法

我们强烈建议你在任何可能的情况下都要使用 `const`。此外有时改用 C++11 推出的 constexpr 更好。

- 如果函数不会修改传你入的引用或指针类型参数，该参数应声明为 `const`。`
- 尽可能将函数声明为 `const`。访问函数应该总是 `const`。其他不会修改任何数据成员，未调用非 `const` 函数，不会返回数据成员非 `const` 指针或引用的函数也应该声明成 `const`。
- 如果数据成员在对象构造之后不再发生变化，可将其定义为 `const`。

## 5.8. `constexpr` 用法

在 C++11 里，用 constexpr 来定义真正的常量，或实现常量初始化。

## 5.9. 整型

C++ 内建整型中，仅使用 `int`。如果程序中需要不同大小的变量，可以使用 `<stdint.h>` 中长度精确的整型，如 int16_t。如果您的变量可能不小于 2^31 (2GiB)，就用 64 位变量比如 int64_t。此外要留意，哪怕您的值并不会超出 int 所能够表示的范围，在计算过程中也可能会溢出。所以拿不准时，干脆用更大的类型。

## 5.10. 64 位下的可移植性

代码应该对 64 位和 32 位系统友好。处理打印，比较，结构体对齐时应切记：

- 注意 `PRI*` 宏会被编译器扩展为独立字符串。因此如果使用非常量的格式化字符串，需要将宏的值而不是宏名插入格式中。使用 `PRI*` 宏同样可以在 `%` 后包含长度指示符。例如，`printf("x = %30"PRIuS"\n", x)` 在 32 位 Linux 上将被展开为 `printf("x = %30" "u" "\n", x)`，编译器当成 `printf("x = %30u\n", x)` 处理 (Yang.Y 注：这在 MSVC 6.0 上行不通，VC 6 编译器不会自动把引号间隔的多个字符串连接一个长字符串)。
- 要非常小心的对待结构体对齐，尤其是要持久化到磁盘上的结构体。
- 创建 64 位常量时使用 LL 或 ULL 作为后缀，如：

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

- 如果你确实需要 32 位和 64 位系统具有不同代码，可以使用 `#ifdef_LP64` 指令来切分 32/64 位代码。 (尽量不要这么做，如果非用不可，尽量使修改局部化)。

## 5.11. 预处理宏

使用宏时要非常谨慎，尽量以内联函数，枚举和常量代替之。
宏意味着你和编译器看到的代码是不同的。这可能会导致异常行为，尤其因为宏具有全局作用域。

## 5.12. 0，nullptr

整数用 `0`，实数用 `0.0`，指针用 `nullptr`，字符 (串) 用 `'\0'`.

## 5.13. sizeof

在能拿到对象实例时，必须用 `sizeof(varname)` 代替 `sizeof(type)`，防止对象类型发生变化导致结构空间计算错误，`sizeof(type)` 只用于处理不涉及对象实例的代码。

## 5.14. auto

在局部作用域中，可以使用 `auto` 绕过烦琐的类型名，增加可读性。禁止在全局（包括对外命名空间）作用域中使用 `auto`。

## 5.15. Lambda 表达式

适当使用 lambda 表达式。禁止默认 lambda 捕获，所有捕获都要显式写出来，防止可能的内存泄露等问题。

**↑ 返回规则等级划分**

# 6. 命名约定

## 6.1. 通用命名规则

函数命名，变量命名，文件命名要有描述
代码中禁止使用拼音和英文混合命名，建议全部使用英文命名。强烈不建议符号以下划线和美元符号开始或结束。

## 6.2. 文件命名

建议文件名和类型名保持一致。例如：`MyClassName.cpp`。

## 6.3. 类型命名

类型名称使用 UpperCamelCase 方式，每个单词首字母均大写，其中不包含下划线，例如：`MyExcitingClass`，`MyExcitingEnum`。

## 6.4. 变量命名

变量 (包括函数参数) 和数据成员名以 `lowerCamelCase` 方式命名。 成员变量可以在前面添加 `m` 来区分是类成员变量，例如：`value`，`myValue`，`mClassFiled`。

## 6.5. 常量命名

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，名称全部大写，单词用下划线分割，例如：`MAX_LEN`。

## 6.6. 函数命名

常规函数使用 `lowerCamelCase` 方式命名，例如：`getValueName()`。

## 6.7. 命名空间命名

多单词命名空间名称使用 `UpperCamelCase` 的方式，每个单词首字母均大写，不包含下划线，单个单词的命名空间名称可以使用小写。例如：`namespace MyUtils {}`，`namespace demo {}`。

## 6.8. 枚举命名

枚举成员的命名应当和 *常量* 一致，名称全部大写，单词用下划线分割，例如：ENUM_NAME。

枚举类型命名应当和 *类型* 命名一致，采用 `UpperCamelCase` 方式。

## 6.9. 宏命名

宏的命令应当和 *常量* 一致，名称全部大写，单词用下划线分割。特别的，对于宏的命名要做到具体，见名知意，不要模棱两可，例如: MY_MACRO_THAT_SCARES_SMALL_CHILDREN。

## 6.10. 目录命名

源码目录建议使用一个单词命名，单词建议小写，统一使用英文单数形式。

**↑ 返回规则等级划分**

# 7. 注释

注释的目的是为了让代码意图容易理解，**写准确明了的注释 ，避免写多余的，错误的注释。**

1. [必须] 接口类，导出函数，必须要写明注释。说明函数作用，参数，返回值的意义。使用 // 或 /* */，注意务必保持统一。
2. [必须] 单行注释使用 //。
3. [建议] 对代码中晦涩难明的部分。使用注释加以阐释。
4. [建议] TODO 注释中，可以写明修复的时机，避免变成没法实现的承诺。可以使用IDE中的TODO注释查询功能，定期查看删除不再需要的注释。
5. [建议] 废弃(deprecated)注释，在C++14以后，可以用 [[deprecated]] 属性代替。
6. [建议] 除了文件头部，不要在注释中写 // Add by ${NAME} 这样的署名。
7. [建议] 避免直接注释代码，如果有特殊的原因没法直接删掉代码，而是临时注释，那么在注释部分写清楚注释代码的原因。

**↑ 返回规则等级划分**

# 8. 格式

## 8.1.基本要求

1. [必须] 单行代码字符数不超过 100， 超出须换行。例外：头文件保护(include guard)， #include 不受此约束。
2. [必须] 源码文件编码需要统一， 跨平台源码文件使用UTF-8编码。
3. [必须] 源码使用4个空格缩进， 禁用制表符。
4. [建议] 函数定义， 函数体起头的 { 不换行。
5. [必须] 对于函数签名定义， 返回类型和函数名在同一行， 如果放不下就对参数分行。
6. [必须] 对于函数调用， 如果参数超长， 对参数分行， 不在逗号前换行. 对于调用链， 不在左括号前换行。

7. [建议] 条件语句块， 使用egyptian brackets方式缩进， 对于单行表达式， 不要省略{}。

## 8.2. Lambda 表达式

Lambda 表达式对形参和函数体的格式化和其他函数一致；捕获列表同理，表项用逗号隔开。

## 8.3. 函数调用

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。

## 8.4. 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，即使只有一行也要用大括号。

```
if (condition) {  // 圆括号里没有空格,且左/右圆括号前后皆有一个空格
  ...  // 2 空格缩进.
} else if (...) {  // else 与 if 的右括号同一行.
  ...
} else {
  ...
}
```

## 8.5. 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`。

## 8.6. 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;
```

## 8.7. 布尔表达式

如果一个布尔表达式超过标准行宽，断行方式要统一一下。
下例中，逻辑与 (`&&`) 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) {
  ...
}
```

注意，上例的逻辑与(`&&`) 操作符均位于行尾.所有操作符放在开头也可以，但需要统一。

## 8.8. 函数返回值

不要在 return 表达式里加上非必须的圆括号。

## 8.9. 预处理指令

预处理指令不要缩进，从行首开始。

## 8.10. 类格式

访问控制块的声明依次序是 `public:` ，`protected:` ，`private:` 。

## 8.11. 构造函数初始值列表

构造函数初始化列表放在同一行或按四格缩进并排多行。

## 8.12. 命名空间格式化

命名空间内容缩进。

```
namespace {

void foo() {  // 正确. 命名空间内没有额外的缩进.
  ...
}

}  // namespace
```

## 8.13. 垂直留白

垂直留白越少越好，建议连续垂直留白至多1行。

## 8.14. 函数长度

函数建议不超过40行。

## 8.15. 文件长度

文件建议不超过800行。

## 8.16. 函数代码块缩进

一个函数内部的代码块缩进建议不超过4层。

```
void foo() {
    for (int i = 0; i < 10; ++i) {
        ...  // 一层缩进。
        for (int j =0; j < 10; ++j) {
            ...  // 二层缩进。
        }
    }
}
```

↑ 返回规则等级划分