Coding Rule

1. 前言

本文适用于开发 iAuto 以及其衍生项目时应该遵守的编程规范。

编程规范对软件开发以及程序员来说非常重要，主要是因为软件的维护费用在整个软件的生存期里占有很大的比例，一般来说软件维护活动所花费的工作占整个生存期工作量的 70%以上，一些大型系统的比例更高。

因此，为了更好的理解源代码，以方便进行软件维护，就需要制定编程规范，在早期编程阶段共同遵守这些规范。

所有的软件开发人员，必须遵守本文中规定的编程规范。

2. 简介

2.1 编程规范的内容

本编程规范侧重代码的行文规范，主要致力于提高代码的一致性和可读性，也是本规范的重中之重，占用了本文 80%以上的篇幅。

另外，本规范中还会强调一部分编程习惯问题，以提高代码的正确性，代码执行效率和开发效率。但是如何提高代码的正确性是一个很大的话题，单独展开就可以写一本书了，所以不作为本规范中的重点内容。

2.2 编程规范的基准

本规范主要基于下面的要求来制定。

2.2.1 代码的一致性

很多人都有自己的编程习惯，比如有人喜欢用空格缩进，有人喜欢用 Tab 缩进；即使都用空格缩进也有不同，有人喜欢用 4 空格，有人喜欢用 2 空格，这就带来代码的不一致性。如果某个文件或者模块都是一个人写还好，如果多人修改过，那代码看起来就乱七八糟了，因为缩进有可能完全混乱了。看下面的这个例子：

```
    for (int i = 0; i < 100; i++) {
        // 这里是 Tab 缩进
```

```
    for (int j = 0; i < 100; j++)
    // 这里是两空格缩
}
```

2.2.2 代码的可读性

好的编程规范能极大的提高代码的可读性，这对强调多人协同开发的大型项目来说尤为重要。代码的协作性表现在：

- 直接协作
  - 即组间协作，比如 A 组要用到 B 组提供的一个模块，如果 A、B 两组编程规范不一致，或者根本没有编程规范，那么 A 组要去消化 B 组提供的那套乱七八糟的代码可能要花很长时间，而且正确性也不能保证。
- 代码 Review
  - 代码 Review 也是很重要的环节，如果代码不规范，可读性极差，那么 Review 的时间可能变得很长，而且 Reviewer 可能会随着极差代码心情变得极差，直接影响 Review 的效果。
- 代码维护
  - 没有太多人喜欢维护的工作，更没有人会把一个模块维护一辈子的，代码维护对象的变更是常有的事。遵守规范的代码会让代码交接工作变得简单，甚至会让后来者从中受益；没有规范的代码只会让人更讨厌维护。

2.2.3 代码的正确性

虽然不是本规范的重点，但是对于常见的正确性问题也做了一些说明。举例来说，C 编程中的一个经典错误可能是这样：

```
if (c = 1) {// 笔误！作者本意是判断 c == 1
    printf("c is equal to 1\n");
}
```

但是如果编程规范中规定，比较操作时，常量一定要放在左值，那代码可能变成：

```
if (1 = c) {// 同样的笔误，但是会被编译器检测到
```

```
    printf("c is equal to 1\n");
}
```

一样的错误，不一样的命运！像这种实用的编程风格，一定要变成每个程序员固有的习惯！

2.2.4 代码的执行效率

虽然很多人都知道 i++和++i 的不同，但是大部分新手还是习惯性的使用 i++。殊不知，这两种写法会带来巨大的效率差异！本规范中涉及执行效率的内容，就是希望能够通过改变开发者一点点的编程习惯，来改善一定的代码执行效率。

2.2.5 提高开发效率

很多不规范的编程习惯甚至会影响到开发效率！比如说，很多不太注意的程序员，会在头文件中随意引用其他的头文件，这将会导致引用此头文件的 cpp 文件编译速度变慢，而且一旦某个头文件改变，可能会导致无数的 cpp 文件重编，这是对开发效率的巨大伤害！

2.3 编程规范指定上的考虑

本编程规范在制定时参考了：

- 之前公司开发项目的编程经验
- 《高质量 C/C++编程》
- Google 开发规范
- 当前流行的开源库，如 Webkit/Chrome/Qt 等

2.4 阅读本规范需要注意的东西

本文中规定的规范，适用于（但不限于）C/C++的编程。有特例的情形，会特别注明。

本文中规定的规范，大体上分为`[必须]`和`[推荐]`两部分。

标记	说明
`[必`	表示在编码过程中必须共通遵守的项目

标记	说明
须]	
[推荐]	表示推荐项目，尽量遵守，在编码过程中不做强制要求
[例外]	为了方便编码，可以不遵守指定规范的的项目

3. 文件规范

3.1 文件命名

这里提到的文件主要分为两类：

- 头文件：通常所说的扩展名是`.h`的文件
- 源文件：有 C 语言和 C++语言之分
  - C 语言的扩展名为`.c`
  - C++语言的扩展名为`.cpp`

3.1.1 基本文件命名规则

原则上一个 Class 对应一组`.h`头文件和`.cpp`源文件，文件名和 Class 名相同。[推荐]

[例外]如果一个头文件中有多个 Class 时，若是下列情况，可以作为例外。

- 一个 Class 作为主 Class，其他 Class 为其服务。（此时主 Class 作为文件名）
  - 建议把其他 Class 定义在主 Class 内部作为嵌套 Class。[推荐]
- 两个 Class 有紧密的联系（此种情形仅限于 Event 通知的定义，继承除外）

判断的基准：这些 Class 必须有一个特点：其中一个被用到时，另外一个也会同时被用到。[推荐]

3.1.2 扩展名

扩展名必须全部使用小写字母。[必须]

- 头文件：

- 扩展名一律为`.h`。`[必须]`

- 源文件：
  - C 的扩展名为`.c`。`[必须]`
  - C++的扩展名为`.cpp`。`[必须]`

3.1.3 文件名

文件名命名规范遵循下面的方式。`[推荐]`

- 含有 Class 的 C++的文件名必须和 Class 名一致，大小写也和 Class 名一致。
- 其他文件命名采用单词首字母大写，单词连写的形式。例如，`Sample.h`, `SampleCode.h`, `SampleCode.cpp`。
- 实际上，在大型项目中组织代码，还经常要在有意义的文件名前，再加入文件的分类。比如，如果 SampleCode.cpp 属于 Network 模块，那文件名可能变为：`NetworkSampleCode.cpp`。
- 除了前缀，文件有可能会被加上后缀。比如要实现 Semaphore 功能，可以基于 Linux 的系统调用，也可以基于 Posix 的 API，这两种实现共存在项目中的时候，可以根据后缀来区分，文件名可能变成这样： `SemaphoreLinux.cpp`, `SemaphorePosix.cpp`。

3.2 文件内的排列顺序

3.2.1 头文件的排列顺序

头文件中按照下面的顺序进行排列。`[必须]`

1. 文件开头注释（参考 12.1.1.1）
2. 头文件的包含保护宏（参考 3.4）
3. 包含的头文件（参考 3.3）
4. 宏定义
5. 常量
6. 其他类型声明
7. C 类型函数声明
8. 类定义（参考 9.1.4）
9. 文件结尾注释（参考 12.1.1.1）

3.2.2 C/C++源文件的排列顺序

源文件中按照下面的顺序进行排列。[必须]

1. 文件开始注释（参考 12.1.1.2）
2. 包含的头文件（参考 3.3）
3. 静态常量
4. 变量的定义
5. 静态函数的声明/定义
6. 类构造函数
7. 类析构函数
8. 其他
9. 文件结尾注释（参考 12.1.1.2）

3.3 头文件包含规范

3.3.1 头文件包含规则

- 使用<>来引用系统头文件。（参考 3.3.2.1）[必须]
- 使用<>来引用外部头文件。（参考 3.3.2.2）[推荐]
- 使用""来引用内部头文件。[必须]

例如，

```
#include <stdlib.h>
#include "other.h"
```

3.3.2 头文件的包含顺序

头文件按照先系统头文件，再外部头文件，最后是内部头文件的形式进行包含。[必须]

3.3.2.1 系统头文件

所谓的“系统头文件”，一般包括：

- C 库的头文件，如<stdio.h>
- 操作系统相关的头文件，如 <sys/stat.h>

- 标准库的头文件，如`<vector>`
- 非标准库（第三方库）的头文件，如 GLib 的`<glib.h>`

引用系统头文件时，注意要按照上面的顺序。[必须]

3.3.2.2 外部头文件

外部头文件是指工程内其他模块提供的外部公开的头文件。

外部头文件要按照模块的逻辑层次，从低到高来引用。[必须]

3.3.2.3 内部头文件

内部头文件是指模块内部的头文件，包含了提供给外部公开的头文件和非公开的头文件。

内部头文件也要按照模块的逻辑层次，从低到高来引用。[必须]

3.3.3 不要包含无用的头文件

不要包含无用的头文件。[必须]

引用无用的头文件，会导致很多问题：

- 大部分的头文件循环引用问题都是滥用头文件导致的
- 在某个 cpp 中引入大量的头文件，会导致编译速度变慢
- 某个头文件有修改，可能会导致项目中无数无关的 cpp 文件重新编译

最容易发生滥用头文件的场合是：

- 不知道该使用哪个头文件，就全部都包含进去
- 曾经使用了某个头文件里的东西，后来不用，但是没有删除

3.3.4 包含头文件时使用相对路径

在头文件中使用相对路径是很危险的行为。

- 路径变更以后，很可能会引起编译错误
- 一些系统头文件的安装路径，在不同的系统中，甚至不同的版本上，都是不一样的，这会带来移植性问题

在包含头文件时，禁止使用像下面一样的相对路径（含有`..`的路径）。[必须]

```
#include "../../folder/Sample.h"
```

如果公开头文件的目录有多级目录时，必需按照下面的方式包含。（参考 iAuto 命名规范）[必须]

```
#include "subfolder/Sample.h"
```

3.3.5 通过显示的类声明避免引入过多的头文件

通过显示的类声明避免引入过多的头文件。[推荐]

在很多场合下，对头文件的引用可能只是为了使用其中的某个类型定义。比如 ClassB.h 用到 ClassA.h 头文件中定义的 ClassA：

```
// ClassB.h
#include “ClassA.h”
class ClassB
{public:
...private:
ClassA *m_classA;
};
```

由于没有用到 ClassA 的具体结构，所以 ClassB.h 其实是没有必要包含 ClassA.h，只要改成如下形式就可以避免：

```
// ClassB.h
class ClassA; // 使用声明ClassA 替代包含ClassA.h
class ClassB {
public:
...private:
ClassA *m_classA;
};
```

3.3.6 包含外部公开的头文件

包含外部头文件时，只能包含公开的头文件。禁止包含其他模块内部非公开的头文件。[必须]

3.4 头文件的包含保护

- 在头文件定义并使用包含保护宏。[必须]
-
```
#ifndef SAMPLE_H  // 多重包含保护
#define SAMPLE_H  // 包含保护宏定义...
#endif /* SAMPLE_H */
```
-
- 第三方开发库的头文件里可能没有使用包含保护，包含这些头文件时，使用扩展的包含保护。[推荐]
-
```
#ifndef MATHLIB_H
#include <mathlib.h>     // 第三方库头文件
#define MATHLIB_H        // 包含保护宏定义
#endif /* MATHLIB_H */
```
- 如果可以确定第三方库的头文件中已经使用了包含保护，不需要再定义包含保护，直接包含头文件。[推荐]
-
```
#include <QWidget>          // Qt 库头文件
```
-

3.4.1 包含保护宏的定义

头文件中需要在开头的位置定义包含保护宏，包含保护宏参照以下形式（假定头文件为Sample.h）。

`SAMPLE_H`

即：文件名所有字母大写，`.h`变为`_H`。`[必须]`

```
#define SAMPLE_H
```

3.4.2 头文件中的包含保护

下面是头文件中使用包含保护的例子。

```
#ifndef SAMPLE_H            // 多重包含保护
#define SAMPLE_H            // 包含保护宏的定义
#include <stdlib.h>         // 标准库头文件
#ifndef MATHLIB_H
#include <mathlib.h>     // 第三方库头文件
#define MATHLIB_H
#endif /* MATHLIB_H */
#include <QWidget>          // Qt 库头文件
#include “Log.h”            // 工程内头文件
...


#endif /* SAMPLE_H */
```

3.4.3 源文件中的包含保护

下面是源文件中使用包含保护的例子。

```
#include <stdlib.h>         // 标准库头文件
#ifndef MATHLIB_H
#include <mathlib.h>     // 第三方库头文件
#define MATHLIB_H
```

```
#endif /* MATHLIB_H */
#include <QWidget>          // Qt 库头文件
#include “Log.h”            // 工程内头文件
...
```

3.5 注明 C++头文件

如果是 C++的头文件，在文件中使用下面的方式注明。[必须]

```
#ifndef __cplusplus
#error ERROR: This file requires C++ compilation (use a .cpp suffix)
#endif
```

3.6 头文件中的其他规则

3.6.1 不要在头文件中定义变量

在头文件中定义变量，会导致变量生成多份拷贝，所以一定要避免。[必须]

C/C++中，所有的“定义”（definition）是相对“声明”(declaration)而言的，区分“定义”和“声明”的唯一标准是是否会造成内存使用。

```
// 声明的例子
extern int a;

struct Type
{
};

// 定义的例子
int b;
Type t;
```

3.6.2 不要在头文件中使用 `using namespace`

在头文件中禁止使用 `using namespace xxx`。[必须]

使用 `namespace` 的目的就是为了避免污染全局命名空间。但是代码中如果用了 `using namespace xxx`，那 `namespace` 本来的意义就没了，所以 `using namespace` 的操作尽量慎用。

如果用在头文件中，因为无法保证这个头文件再会被谁使用，这就造成了 namespace 完全暴露给了所有包含了此头文件的 cpp，这会造成严重的命名空间污染。

在 `cpp` 文件中，如果保证不会有全局命名空间污染的问题，那么可以使用 `using namespace xxx`。

4. 缩进与换行

4.1 缩进的格式

由于很多不同的编辑器处理 Tab 和空格的方式不一样，会导致排版的混乱，因此要求以 4 个空格取代 Tab。

- 在编码中禁止使用 Tab，使用空格取代。[必须]
- 使用 4 个空格进行缩进。[必须]

4.2 避免长行

如果代码行过长，也会影响代码的可读性，所以有必要限制每行的最大字符数。

建议每个代码行不要超过 100 个字符。[推荐]

4.3 函数的换行

4.3.1 函数中的换行要以参数为起点

函数中的换行必须要以参数为起点。[必须]

正确的例子：

```
unsigned longSomeClass::hook(int anArg, Object anotherArg,
        String yetAnotherArg, Object andStillAnother);
```

不合适的例子：（声明/定义时从参数中间断行）

```
unsigned longSomeClass::hook(int anArg, Object
    anotherArg, String yetAnotherArg, Object andStillAnother);
```

又一个不合适的例子：(调用函数从某个参数中间断行)

```
instance.hook(getBigness(10,
    20, 30),
    A, "xxx", B);
```

一种正确的写法：

```
instance.hook(getBigness(10, 20, 30),
              A, "xxx", B);
```

4.3.2 新行的缩进方式

新行可以有两种缩进方式：[必须]

- 相对首行缩进 4 个空格
- 新行从首行左括号处开始缩进

选择缩进方式的原则是（以代码行不超过 100 个字符为基准）：[推荐]

- 如果函数名比较短，左括号位置比较靠前，建议从对齐左括号的地方开始缩进。
- 如果函数名比较长，左括号位置比较靠后，建议以相对首行 4 个字符的方式开始缩进。

缩进换行的例子：

-

函数名比较短，以对齐左括号的方式换行：

-

```
unsigned int xxx(int anArg, Object anotherArg,
                 String yetAnotherArg, Object andStillAnother);
```

-

-

  函数名比较长，以缩进 4 个空格的方式换行：

-

```
unsigned int longSomeClass::hook(int anArg, Object anotherArg,
    String yetAnotherArg, Object andStillAnother);
```

-

-

  无序的缩进换行，应该禁止：

-

```
unsigned longSomeClass::hook(int anArg, Object anotherArg,
                 String yetAnotherArg, Object andStillAnother);
```

-

4.3.3 函数体的换行

函数体定义需要遵循下面的规范：

-

  {和}分别单独占一行。[必须]

-

```
void doSomething()
{
```

```
    ...
}
```

- [例外]如果是定义在头文件中的空实现，可以不遵守上述规范。此时，需将{和}写在一起，并放置在函数后面，和)之间留一个空格。[必须]通常会在空实现的虚析构、callback 类中使用。

```
class SampleCallback
{
public:
    virtual ~SampleCallback() {}

    virtual void callback1() {}
    virtual void callback2() {}
}
```

-

- 一行中最多有一条语句。[必须]

-

```
doSomething(); doAnotherThing(); // 不可以！!!
```

-

4.4 表达式中的换行

4.4.1 条件表达式中的换行

条件表达式中如果出现较长的表达式，需要进行换行时，按照下面规则换行。

- 以二元操作符为起点进行换行。[必须]
- 新行相对缩进 4 个空格。[必须]

正确的例子：

```
if ((condition1 && condition2)
    || (condition3 && condition4)
    || (!(condition5 && condition6))) {
    doSomethingAboutIt();
}
```

4.4.2 含运算符的赋值语句的换行

赋值语句中如果出现较长的运算符表达式，需要进行换行时，按照下面规则换行。

- 以运算符为起点进行换行。[必须]
- 新行可以采用下面两种缩进方式。[必须]
  - 新行跟=后第一个操作数对齐
  - 新行缩进 4 个空格

正确的例子：

```
longName1 = longName2 * (longName3 + longName4 - longName5)
        + 4 * longName6;


// 换行后新行内容过长，缩进4 个空格 longName1 = longName2 * (longName3 + longName4
- longName5)
    + longName6 * (longName7 + longName8 + longName9) + longName10 - longName11;


// 也可以多次换行 longName1 = longName2 * (longName3 + longName4 - longName5)
            + longName6 * (longName7 + longName8 + longName9)
            + longName10 - longName11;
```

如果新行过长（比如超过 100 个字符），采用缩进 4 个空格的方式。[推荐]

4.5 含有成员的定义中的换行

像 `struct`，`enum`，`union`，`class` 等含有成员的定义，需要遵循下面的规范：[必须]

- {单独占一行

- }和;写在一起，并单独占一行
- 每个成员单独占一行

定义的例子：

```
enum MyEnumeration
{
    EnumValue1,
    EnumValue2,
    ...
};

struct MyStruct// MyEnumeration 和MyStruct 定义之间用空行分开。
{
    char*   name;
    int32_t length;
};

union MyUnion
{
    void*   addr;
    int32_t intValue;
};

class MyClass
{
public:
    MyClass();
    ~MyClass();

    int32_t width();
```

```
    void setWidth(int32_t value);

private:
    int32_t m_width;
};
```

如果使用 C 类型的 `struct`，`enum`，`union`（有 typedef），为了遵守上面的规范，可以写成下面的形式。

```
enum tagMyEnumeration
{
    EnumValue1,
    EnumValue2,
    ...
};
typedef enum tagMyEnumeration MyEnumeration;

struct tagMyStruct// MyEnumeration 和MyStruct 定义之间用空行分开。
{
    char*  name;
    int32_t length;
};
typedef struct tagMyStruct MyStruct;

union tagMyUnion
{
    void*  addr;
    int32_t intValue;
};
```

```
typedef union tagMyUnion MyUnion;
```

5. 空行与空格

代码中合适的空格与空行能显著的提升代码可读性，也是影响到代码质量的关键因素之一。

5.1 用空行对代码“分区”

代码中有很多区域，比如头文件中，可能会有对其他头文件的引用，有宏定义，也有类定义等，使用空行可以很自然的把这些区域给隔离开来。[推荐]

```
// 第一分区
#ifndef SAMPLE_H
#define SAMPLE_H
// 第二分区，头文件引用区
#include <stdlib.h>
// 第三分区，宏定义区
#define MAX_COUNT 10
// 第四分区，相关类声明区
class other;

// 第五分区，类声明区
class MyWidget : public QWidget
{
public:
    ...
};

// 第六分区
#endif /* SAMPLE_H */ /* EOF */
```

5.2 用空行对代码“分组”

并不是每个区域前后分别只要一个空行就行了，在比较大的区域内，还需要根据一定的情况，用空行把代码分成“组”。以下介绍一些分组的基本原则。

5.2.1 根据头文件类型的不同分组

3.3 中提到了头文件几个类型。如果代码中需要引用大量的头文件，可以考虑根据头文件的不同类型进行分组。[推荐]

5.2.2 根据功能分组

在同一个区域内，最好能根据功能块进行分组。比如，

- 在函数声明区域，可以把有联系的函数声明放在一起，用空行与其他部分隔开。[推荐]
- 在宏定义区域，把实现相关的宏定义写在一起，用空行与其他部分隔开。[推荐]

下面几个函数声明，分别涉及到文件操作和文件系统操作，可以用空行分隔：

```
// 文件系统操作相关的函数
int PFA_remove(const char* inPath);
int PFA_rename(const char* oldname, const char *newname);

// 文件操作相关的函数
int PFA_open(const char *pathname, int mode, ...);
int PFA_close(int fp);
size_t PFA_read(int fp, void* buffer, size_t count);
size_t PFA_write(int fp, const void* buffer, size_t count);
```

下面的四个宏定义分别实现了两个不同的功能，可以用空行分隔：

```
#define WSTR__(x) L ## x #define WSTR(x) WSTR__(x)
#define M2WSTR(x) WSTR(#x) #define M2STR(x) #x
```

5.2.3 类型定义

`struct`，`union`，`enum`，`class`类型定义后应该用空行隔开。（参考 4.5 中的例子）[必须]

5.2.4 函数定义

函数定义后，应该用空行隔开。[必须]

```
void method1()
{
    ...
}

void method2() // method1()和method2()之间用空行隔开。
{
    ...
}
```

5.3 用空行分隔逻辑块

在一个函数内部，也要根据逻辑的不同，用空行分隔代码，增加代码可读性。[推荐]

以下是一个典型的函数内部实现的例子：

```
BookmarkInfo* BookmarkManager::getAt(uint32_t index)
{
    // 逻辑块1：类型定义
    BookMarkURLElement bookMarkElement(L"", L"");

    // 逻辑块2：判断index是否在范围内
    if (eResult_ok != getURLItemAt(index, bookMarkElement)) {
        return NULL;
    }

    // 逻辑块3：创建对性并赋值
```

```
    BookmarkInfo *info = new BookmarkInfo;
    info->url = bookMarkElement.getUrl();
    info->title = bookMarkElement.getTitle();

    // 逻辑块4：返回
    return info;
}
```

5.4 类定义内部的空行

5.4.1 不同区域间的空行

public/protected/private 区域间要有空行。（参考 5.4.5）[必须]

5.4.2 功能分组

- 函数声明区域内，要根据功能的不同进行分组。（参考 5.4.5）[推荐]
  - 可以所有的构造/析构函数写在一起，后面用空行分隔。
  - 可以按功能把函数分组，前后用空行隔开。
- 变量定义区域内，要根据功能的不同进行分组。（参考 5.4.5）[推荐]
  - 如果类成员变量数量很多的时候，使用空行分组能显著提高代码可读性。
  - 分组的原则与函数类似，按照功能进行分组。

5.4.3 不同类别成员

在相同的 public/protected/private 区域内，不同类别成员之间需要有空行分隔。（参考 5.4.5）[必须]

类成员分为以下类别：

- 常量定义
- 自定义类型（`enum/union/struct/class`等）
- 静态成员变量
- 成员变量
- 构造函数/析构函数

- 操作符重载函数
- 多态函数
- 专有函数

5.4.4 函数之间空行

函数的注释说明前，必需和前面的内容用空行分隔。（参考 5.4.5）[必须]

[例外]如果是区域内第一个函数，则不需留空行。[推荐]

5.4.5 类定义分组的例子

```
class SampleClass
{
public:
    /// constructor
    SampleClass();
    ~SampleClass();
    SampleClass(const SampleClass& rhs);

    int width();
    void setWidth(int value);

    /// get the height.
    int height();

    /**
     * set the height.
     *
     * @param [IN] value: the height value
     *
     * @return void
     */
```

```
    void setHeight(int value);

private: // private 之前为了和public 区域区分，保留一个空行
    static int totalCount;

    int m_height;
    int m_width;
};
```

5.5 空格的使用

5.5.1 用空格突出关键字

- `const`、`virtual`、`inline`、`case`等关键字之后要留一个空格。[必须]
- `if`、`for`、`while`等关键字之后留一个空格再跟左括号`(`。[必须]
- `do...while`语句中，`do`关键字之后留一个空格再跟大括号`{`。[必须]
- 与关键字不同，函数名之后不要留空格，紧跟左括号`(`。[必须]

5.5.2 括号

- 左括号`(`之后不要留空格。[必须]
- 右括号`)`之前不要留空格。[必须]
- 右括号`)`和左大括号`{`之间留一个空格。[必须]
- 左大括号`{`之后如果有内容，请保留一个空格。[必须]
- 右大括号`}`之前如果有内容，请保留一个空格。[必须]
- 如果左大括号`{`和右大括号`}`出现在一行，并且之间没有内容，请写在一起，不要留空格。[必须]

5.5.3 逗号和分号

- `,`之前不要留空格，如`Function(x, y, z)`。[必须]
- `,`之后要留空格（除非出现在代码行结尾），如`Function(x, y, z)`。[必须]
- 可执行语句之后紧跟`;`，之间不要留空格或其他符号。[必须]

- ;之后有可执行的内容时，其后要留空格。[必须]

;的例子：

```
for (initialization; condition; update) {
    ...
}

// 没内容时写成
for (;;) {
    ...
}
```

5.5.4 二元操作符

二元操作符前后要留空格，如= += >= <= + * % && || << ^等二元操作符的前后应当加空格。[必须]

主要的二元操作符有：赋值操作符、比较操作符、算术操作符、逻辑操作符、位域操作符。

5.5.5 一元操作符

- 一元操作符如! ~ ++ -- &（地址运算符）等与操作数之间不加空格。[必须]
- []的[的前后，]的前面均不加空格。[必须]
- . ->这类操作符前后不加空格。[必须]

5.5.6 不要冗余的空格

为了代码行紧凑起见，不要过多的使用空格。[推荐]

5.5.7 例子

-

函数

-

```
void func1(int x, int y, int z);    // 良好的风格
void func1 (int x,int y,int z);    // 不良的风格
```

-

-

  if 语句

-

```
if (year >= 2000) {          // 良好的风格
    ...
}
if(year>=2000) {             // 不良的风格
    ...
}
if(a>=b&&c<=d) {             // 不良的风格
    ...
}
```

-

-

  for 语句

-

```
for (i = 0; i < 10; i++) {    // 良好的风格
    ...
}
for(i=0;i<10;i++) {           // 不良的风格
    ...
}
for ( i = 0; i < 10; i ++ ) {// 过多的空格
```

```
    ...
}
```

-

-

  `while`语句

-

```
// 良好的风格
while (condition) {
    ...
}
```

-

-

  `do...while`语句

-

```
// 良好的风格
do {
    ...
} while (condition);
```

-

-

  其他

-

```
x = (a < b) ? a : b;          // 良好的风格
x=a<b?a:b;                    // 不好的风格
int *x = &y;                  // 良好的风格
int * x = & y;                // 不良的风格
array[5] = 0;                 // 不要写成 array [ 5 ] = 0;
```

```
a.Function();                // 不要写成 a . Function();
b->Function();               // 不要写成 b -> Function();
```

-

5.5.8 注释中的空格

为了注释能够明晰，便于阅读，注释中空格的使用有以下要求：

- 语句行末的注释之前，至少保留一个空格。[必须]
- 注释标示符与注释内容之间，至少保留一个空格。[必须]
- 行末注释不允许有换行。[必须]

注释中空格使用的例子：

```
statement; // This is a comment.
statement; /* This is a comment. */      <-- 行末注释不允许有换行


// this is a comment. /* this is a comment. */
```

6. 代码

6.1 每条语句一个逻辑

代码行的逻辑要尽可能的简单，不要一条语句行多个逻辑。[推荐]

6.1.1 条件表达时中的逻辑判断

6.1.1.1 条件表达式中的赋值操作

禁止在条件表达式中使用赋值语句：[必须]

```
if (isConnected = network->isConnected()) {// 不可以！禁止在条件表达式中使用赋值语句
    ...
}


// 正确的写法：
```

```
bool isConnected = network->isConnected();
if (isConnected) {
    ...
}
```

6.1.1.2 条件表达式中的函数调用

建议不要条件表达式中进行函数调用。[推荐]

例外除非这个函数比较简单，例如检查状态等。

```
// 不好的例子：
if (VOLUME1 == GetVolume(param1, param2, param3)) {
    ...
}

// 好的例子：
int vol = GetVolume(param1, param2, param3);
if (VOLUME1 == vol) {
    ...
}

// 简单函数调用的例子：
if (network->isConnected()) {
    ...
}
```

6.1.1.3 条件表达式中的算数运算

不要条件表达式中进行算数运算。[推荐]

```
// 不好的例子：
if (MAX_LEN < (length1 + length2)) {
    ...
```

```
}


// 好的例子：
int length = length1 + length2;
if (MAX_LEN < length) {
    ...
}
```

6.1.2 函数调用

不要在函数调用中使用函数调用来替换实参。[推荐]

```
// 不好的例子：
SetSystemVolume(GetVolume(param1, param2, param3));


// 好的例子：
int vol = GetVolume(param1, param2, param3);
SetSystemVolume(vol);
```

6.2 括号

6.2.1 用括号避免操作符优先级问题

为了避免逻辑操作符优先级问题，在每个逻辑操作（包含操作符及其操作数）的两端加上()。[必须]

很多 Bug 的产生是由于逻辑操作符优先级导致的，比如判断整数 a 的 bit0 是否为 1 的表达式：

```
// 错误！"=="优先级大于"&"
if (1 == a&1) {
    ...
}
```

```
// 加括号以后
if (1 == (a & 1)) {
    ...
}
```

在类似的情况下，使用括号来规避错误。

6.2.2 在宏中使用括号

宏函数展开的时候，只是简单的按照宏函数定义的规则，用宏函数中的参数进行展开。

如果请在宏函数中，每个参数的前后都加上括号。[必须]

看下面的例子：

```
#define mul(x, y)    x * y
int a = 3;
int b = 3;
int c = mul(a, b + 1); // 这里展开以后，其实是 a*b+1
printf("%d", c);       // 期待 12，实际输出 10
// 正确的例子：
#define mul(x, y)    ((x) * (y))
```

6.3 比较

6.3.1 整数类型与常量比较

整形与常量比较时，把常量放在前面。[推荐]

这样做的主要目的是避免把==错误的写成=（实际上，这是经常会发生的事情）。

```
if (b = 0) {// 本意是 if(b == 0)
    ...
}

// 如果常量放在前面：
```

```
if (0 = b) { // 本意是 if(0 == b)
    ...
}
// 以上写法会报一个编译错误。
```

6.3.2 指针类型与常量比较

这个与整形类似，需要把常量指针放在前面。[推荐]

这里的常量指针最常用的是 NULL，但是还有一些其他的常量指针，比如 this，常量字符串等。

```
// 和 NULL 比较
if (NULL == name) {
    ...
}

// 和 this 字符串比较
if (this == instance) {
    ...
}
```

6.3.3 避免常量字符串的直接比较

不要直接用==判断字符串是否相等。[必须]

通常情况下，完全相同的常量字符串会被统一化，比如：

```
char *str1 = "abc";
char *str2 = "abc";

printf("%d\n", (str1 == str2)); // 错误！返回 1
```

但是这样做是不可靠的，比如如果两个字符串来自不同的动态库，那个字符串的地址就会不一样。

6.3.4 布尔类型的比较

布尔类型的比较，不要使用`==`或者`!=`，直接使用或者`!`即可。`[必须]`

```
if (false == b) {// 不要这么用
    ...
}


if (!b) {// 正确的做法
    ...
}
```

6.3.5 浮点数的比较

无论是`float`还是`double`类型的变量，都有精度限制。所以一定要避免将浮点变量`==`或`!=`与数字进行比较。`[必须]`

一般的项目中可能会提供一个浮点数与 0 比较的宏，比如：

```
#define EPSILON 0.00001
#define IsFloatZero(x) (((x) >= -EPSILON) && ((x) <= EPSILON))
void SomeMethod(float a)
{
    if (IsFloatZero(a)) {
        ...
    }
    ...
}
```

6.4 常量

6.4.1 避免常数

在代码中，不要直接使用常数（0 除外），应该把常数定义成常量，在代码中直接用常量来表示。[必须]

-

没有注释的常量出现在代码中，很难让人理解。

-

```
err = SomeFunc();
if (-1 == err) {// -1 代表什么意义？
    ...
}
else if (-2 == err) {// -2 代表什么意义？
    ...
}
else {
    ...
}
```

-
-

方便在需求变更时对代码做修改。下面的这个例子里出现了三处常量：

-

```
static char array[100] = { 0 };


void initArry()
{
    for (i = 0; i < 99; i++) {
        array[i] = 'a';
    }
}
```

```
int checkCharCount(char a)
{
    int count = 0;
    int i = 0;

    for (i = 0; i < 100; i++) {
        if (a == array[i]) {
            ++count;
        }
    }

    return count;
}
```

- 这个时候，如果需求有改变，要求数组的大小为 200，那么就至少有三处要修改。

6.4.2 使用正确的常数

C 语言中有一些常用的常数，比如指针的 `NULL`，字符结尾的`'\0'`等，请在合适的时候使用这些常数，不要偷梁换柱。`[必须]`

```
if (!ptr)                   // 不好的做法
if (ptr == NULL)            // 不好的做法
if (0 == ptr)               // 不好的做法
if (NULL == ptr)            // 正确的做法
const int32_t MAX_STR_LEN = 100;
char name[MAX_STR_LEN] = { 0 };

strncpy(name, someone.name, MAX_STR_LEN - 1);
// 下面的语句确保 name 有一个字符串结尾字符
name[MAX_STR_LEN - 1] = 0;      // 不好的做法
name[MAX_STR_LEN - 1] = '\0';   // 正确的做法
```

6.5 变量

6.5.1 每行只声明一个变量

每个变量占用一行，否则会降低代码的可读性。[必须]

```
// 不好的例子
int left, right, *top;
// 改进后的例子：
int left = 0;
int right = 0;
int* top = NULL;
```

6.5.2 变量在定义时初始化

很多 Bug 的产生都与变量未初始化有很大的关系，所以在定义变量时对变量初始化。[必须]

以下几种类型的初始化需要特别注意一下：

- 指针类型：如果没有特定的值用于初始化，请务必初始化为 NULL
- 枚举类型：一般枚举类型在声明时，会在末尾增加一个无效的枚举项，这个枚举项可以用来初始化
- 数组类型，可以在定义时直接赋值为 0，比如：

```
char buffer[128] = { 0 };
```

-
- 引用类型必须在定义时初始化（引用类型的类成员变量在初始化列表进行初始化），但是切忌初始化为空引用：

```
// 切忌空引用！！！
int &a = *(int*)(0);
```

-

6.5.3 C 类型结构体初始化

C 类型的结构体类型：C 类型的结构体类型无法在定义时加入构造函数，一般可以在定义变量的时候直接赋值，或者在定义以后用 `memset` 初始化。`[推荐]`

```
struct tagSize
{
    int width;
    int height;
};
typedef tagSize Size;
// 初始化方式1：定义时赋值
Size a = {100, 100};

// 初始化方式2：memset
Size b;
memset(&b, 0, sizeof(b));

// 初始化方式3：直接赋值
Size c = b;

// 初始化方式4：memcpy，不推荐
Size d;
memcpy(&d, &c, sizeof(Size));
```

6.5.4 C++类型的结构体与类的初始化

在 C++中，类与结构体除了默认的访问权限外，没有区别。对待 C++中的结构体，应该像对待类一样进行处理。

- 初始化 C++的类和结构体，最好办法是定义构造函数。`[推荐]`
- 如果结构体中有类成员时
  - 必须使用 `class` 来定义。`[必须]`
  - 除构造函数外，不能定义任何成员函数。`[必须]`

- 禁止使用 `memset/memcpy` 对类进行初始化。`[必须]`

  例子：

```
// 错误！因为有std::string 类型成员，不能定义成struct。
struct Student
{
    std::string name;
    int id;

    Student();
};
// 正确的方法：
class Student
{
public:
    std::string name;
    int id;

    Student();
};

// 使用构造，运行
Student a();

// 错误！通过memset 初始化，会破坏std::string 类内部的结构
Student b;
memset(&b, 0, sizeof(b));

// 直接赋值，允许
```

```
Student c = a;


// 错误！通过memcpy 初始化，会破坏 std::string 类内部的结构
Student d;
memcpy(&d, &c, sizeof(Size));
```

6.5.5 避免变量名覆盖

变量名覆盖是一定要在代码中避免的！因为覆盖很容易引起混淆。[必须]

以下是几个典型的覆盖的例子：

-

作用域覆盖：

-

```
int count;
...
MyMethod()
{
    if (condition) {
        int count;    // 错误！count 变量名覆盖
        ...
    }
    ...
}
```

-
-

子类成员变量覆盖父类成员变量：

-

```
class Base
{
```

```
protected:
    m_value;
};

class Derived : public Base
{
protected:
    m_value;    // 错误！父类中的m_value 被覆盖
};
```

-

6.6 预处理指令

预处理指令开始的#号必须始终处于一行的行首。[必须]

The hash mark that starts a preprocessor directive should always be at the beginning of the line.

```
// 好的例子：预处理指令从行首开始
    if (lopsided_score) {
#if DISASTER_PENDING      // 正确！从行首开始
        DropEverything();
#   if NOTIFY             // 也可以，#号后空格不是必需的
        NotifyClient();
#   endif
#endif
        BackToNormal();
    }
// 不好的例子：有缩进的预处理指令
    if (lopsided_score) {
        #if DISASTER_PENDING  // 错误！"#if"必须位于行首
```

```
DropEverything();
        #endif                  // 错误！不要缩进"#endif"
        BackToNormal();
    }
```

7. 控制流

7.1 `if`的写法

`if/else`写法的基本规则：`[必须]`

- `if/else if/else`分别独占一行
- `if/else if/else`必须有{}
- {和`if/else if/else`语句写在同一行
- {前留一个空格
- }独占一行
- 有`else if`的时候，最后的`else`不可省略

以下是一个不加{}导致错误的例子：

```
  if (1 == a)
      if (1 == b) {
      }
  else if (2 == a)  // 这个else 与谁匹配？！
      ...
```

以下是正确的写法：

```
  if (1 == a) {
      if (1 == b) {
      }
  }
  else if (2 == a) {
  }
```

```
else {
}
```

7.2 while的写法

while写法的基本规则与if/else基本相同。[必须]

- while必须有{}
- {和while语句写在同一行
- {前留一个空格
- }独占一行

while的例子：

```
while (condition) {
    ...
}
```

7.3 do...while的写法

do...while写法的基本规则：[必须]

- do/while分别独占一行
- do...while必须有{}
- {和do语句写在同一行
- {和do之间留一个空格
- }和while语句写在同一行
- }和while之间留一个空格

do...while的例子：

```
do {
    ...
} while (condition);
```

7.4 switch/case的写法

`switch/case` 写法的基本规则：`[必须]`

- `{`与 `switch` 语句写在同一行
- `{`前留一个空格
- `}`独占一行
- `case` 不需要缩进（避免过多的缩进）
- 推荐在 `case` 中使用`{}`
- 必须有 `default`
- 如果某个 `case` 没有 `break`，要注释说明
- `default` 不能省略
- `default` 无论是否有内容，`break` 不能省略

`switch/case` 的例子：

```
switch (condition) {
case 1: // case 不需要缩进
    {
        statements;
    }
    // 有内容，又不需要 break 的地方，在这里要加入注释
case 2:
    {
        statements;
    }
    break;
case 3: // 无内容，又不需要 break 的地方，在这里加入注释
case 4:
    {
        statements;
    }
    break;
default: // default 分支不能省略
```

```
    {
        statements;
    }
    break; // default 的break 语句不能省略
}
```

7.5 for的写法

for写法的基本规则：[必须]

- for必须有{}
- {和for语句写在同一行
- {前留一个空格
- }独占一行

for的例子：

```
for (initialization; condition; update) {
    ...
}
```

8. 命名规范

8.1 命名规则

代码中遵循下面的命名规则，在同一个类，或者同一文件中，相同类别的命名必须保持一致的命名规则。[必须]

相同的机能模块建议采用相同的命名规则。[推荐]

8.1.1 骆驼命名法

骆驼式命令法，正如它的名称所表示的那样，是指混合使用大小写字母来构成变量和函数的名字。骆驼命名法的几个特点：

- 名字中不同的单词之间不需要任何分隔
- 从第二个单词开始，每个单词的首字母大写，其余字母小写

※注 1：部分情形下，第一个单词首字母也大写。（参考 8.2）

※注 2：骆驼命名法一般不要求变量根据类型增加前缀。

※注 3：加数据类型前缀的变量命名法，要求变量类型前缀字母全小写。（也满足骆驼命名法）

8.1.2 变量命名规则

变量命名遵循下面的命名规则。

类别	命名规则
临时变量	`fileName`：骆驼命名法[推荐] `strFileName`：加变量类型前缀的骆驼命名法 `file_name`：所有字母小写，单词之间用_分隔
结构体/联合体成员变量	`fileName`：骆驼命名法[推荐] `strFileName`：加变量类型前缀的骆驼命名法 `file_name`：所有字母小写，单词之间用_分隔 ※仅限于 C 库中的函数：扩展名为`.c`源文件及其对应的`.h`文件
类成员变量	`m_fileName`：`m_`+骆驼命名法[推荐] `s_strFileName`：`s_`+加变量类型前缀的骆驼命名法
类静态成员变量	`m_FileName`：`m_`+首字母大写的骆驼命名法[推荐] `m_sFileName`：`m_s`+首字母大写的骆驼命名法 `s_strFileName`：`s_`+加变量类型前缀的骆驼命名法
静态变量	`FileName`：首字母大写的骆驼命名法[推荐] `s_strFileName`：`s_`+加变量类型前缀的骆驼命名法 `s_file_name`：`s_`+所有字母小写，单词之间用_分隔

类别	命名规则
	※仅限于 C 库中的函数：扩展名为`.c`源文件及其对应的`.h`文件
全局变量（不推荐使用）	`FileName`：首字母大写的骆驼命名法`[推荐]` `g_strFileName`：`g_`+加变量类型前缀的骆驼命名法 `g_file_name`：`g_`+所有字母小写，单词之间用`_`分隔 ※仅限于 C 库中的函数：扩展名为`.c`源文件及其对应的`.h`文件

8.1.3 函数命名规则

函数命名遵循下面的命名规则。

类别	命名规则
类成员函数	`setFileName`：骆驼命名法`[推荐]` `SetFileName`：首字母大写的骆驼命名法
非类成员函数	`SetFileName`：首字母大写的骆驼命名法 `set_file_name`：所有字母小写，单词之间用`_`分隔 ※仅限于 C 库中的函数：扩展名为`.c`源文件及其对应的`.h`文件

8.1.4 其他命名规则

其他的一些命名遵循下面的命名规则。

类别	命名规则
类/结构体/联合体	`DefinedType`：首字母大写的骆驼命名法`[推荐]` `defined_type`：所有字母小写，单词之间用`_`分隔

类别	命名规则
	※仅限于 C 库中的函数：扩展名为`.c`源文件及其对应的`.h`文件
枚举	`DeviceType`：首字母大写的骆驼命名法[推荐] `DEVICE_TYPE`：所有字母全大写，单词之间用下划线`_`分隔 `device_type`：所有字母小写，单词之间用`_`分隔 ※仅限于 C 库中的函数：扩展名为`.c`源文件及其对应的`.h`文件
枚举值	对应上面的两种定义方式： `DeviceType_SdCard`：枚举类型名+首字母大写的骆驼命名法，两者之间用下划线`_`分隔 `DEVICE_TYPE_SD_CARD`：枚举类型名+`_`+所有字母全大写，单词之间用下划线`_`分隔 `device_type_sd_card`：所有字母小写，单词之间用`_`分隔 ※仅限于 C 库中的函数：扩展名为`.c`源文件及其对应的`.h`文件
常量	每个单词所有字母都大写，单词之间用下划线`_`分隔

※注：有的项目或者模块会要求在类型定义之前追加前缀，例如，`NI`，`SRCH_`等。这些前缀不受本规范限制，但是除去前缀部分后必须满足上述命名规则。

8.2 名副其实

8.2.1 使用有意义的单词命名

变量的命名应该能够忠实的表达他的意思。

```
void MyCopy(type& a, type& b);          // 不好的命名
```

```
void MyCopy(type& dst, type& src);      // 更有意义的命名
```

8.2.2 不要任意对单词缩写

单词过长的时候，一般会对单词进行缩写。以下是几个缩写的例子：

`error`->`err`, `message`->`msg`。

但是随意的缩写会影响到代码的可读性。以下是几个很恶劣的缩写：

`timeout`->`to`, `implementation`->`imp`

这样的缩写太容易引起歧义，如果不能让人从缩写中迅速还原原始的单词意义，那么这个缩写就是失败的。

8.3 函数命名

8.3.1 自然语言原则（动词+名词）

函数的命名，采用动词+名词的形式更能符合自然语言的语法，也更容易理解。比如，某个函数实现了把某块内存清 0 的功能，那个这个函数应该叫做 `ZeroMemory`，而不是 `MemoryZero`。

但是也有一些场合，采用名词+动词的方式可能会更加合适。比如，要用 C 语言实现一组内存操作的接口：

```
// 接近自然语言的命名方式
void SetMem(void* mem, int value, int len);
void CopyMem(void* dst, void* src, int len);


// 名词+动词的方式：
void MemSet(void* mem, int value, int len);
void MemCopy(void* dst, void* src, int len);
```

显然，在这种场合下，使用名词+动词的方式显然更合适。

一个比较折中的原则是：用 C 语言实现一组功能相似的函数时，采用名词+动词的命名方式，其他情况下，请采用自然语言的方式。

8.3.2 配对命名

如果两个函数具有相关性，那么命名最好能够互相匹配。常用的配对命名：

- Add/Remove
- Insert/Delete
- Start/Stop
- Begin/End
- Send/Receive
- First/Last
- Get/Release
- Put/Get
- Up/Down
- Show/Hide
- Source/Target
- Open/Close
- Source/Destination
- Increment/Decrement
- Lock/Unlock
- Old/New
- Next/Previous

8.3.3 特殊的配对：get/set

get/set 配对时，一般可以把 get 省略掉。比如说：

```
class Sample
{
public:
    int width();
    void setWidth(int value);

private:
```

```
    int m_width;
};
```

8.3.4 特殊的配对：bool 类型

bool 类型一般使用 is/set 配对。

```
class Sample
{
public:
    bool isEnabled();
    void setEnabled(bool value);

private:
    bool m_enabled;
};
```

8.3.5 避免与系统函数重名

与系统函数重名导致的 Bug 隐蔽性很强，在编码阶段就要极力避免这种情况的发生，这就要求程序员要尽可能多的了解系统函数。

- C 库函数

C 库函数虽多，但是由于很常用，记住并不是很困难。

- Linux 系统函数

Linux API 的命名法很特殊，一般采用小写单词+下划线分隔的方式，与骆驼命名法差异很大，一般不容易重名。

- Windows 系统函数

Windows 系统函数大都使用 C 类型函数，命名方法与骆驼命名法是一样的，而且数量极多，特别是短函数，很容易重名。比如 SetRectEmpty 等。避免与 Windows 系统函数重名，尽量少用 C 类型的函数是最有效的办法；如果一定要，而且拿不准的，不妨查一下 MSDN。

8.3.6 避免太简单的函数名和类名

如果没有 namespace 限制的话，全局函数名和类名是直接暴露在全局命名空间中的。如果使用一些非常通用的名词或者动词来命名的话，很容易造成冲突。

比如说，有一个模块用来实现字幕的显示，定义个一个类 `Rect` 来表示每条字幕在屏幕上的位置。在把字幕模块和视频播放模块进行集成的时候，很不幸，视频播放模块也定义一个类 `Rect` 来表示视频在屏幕上的位置，那么这两个同名的 Rect 就会引起混淆。

避免这类冲突的最好办法，就是避免使用这些简单词汇来命名。如果的确是需要用，那么请加上 namespace。

9. 类的规范

9.1 类成员变量/函数定义规范

9.1.1 不要依赖于默认的访问权限

请把所有的成员变量/函数都显示的放到合适的 `public`/`protected`/`private` 区块内，不能依赖默认的访问权限。`[必须]`

```
class SampleClass
{
    int someMethod();    // 错误！需要加上合适的限定符
    ...
};
```

下面的写法是正确的：

```
class SampleClass
{
private:
    int someMethod();    // 正确，说明 someMethod 是一个 private 方法
    ...
};
```

9.1.2 不要定义 public 类型的成员变量

在 `class` 中不要定义 public 类型的成员变量。[必须]

[例外]如果某些类的重要功能是数据容器，可以定义 `public` 类型的成员变量，不过这样的类里除构造函数外，不能定义任何成员函数。（参考 6.5.4）

定义 `public` 类型的成员变量会破坏类的封装性。一般来说，可以定义一组 `public` 的 `get/set` 方法来访问私有的成员变量。

9.1.3 和类关系密切的类型，请定义在类的内部

定义在类内部的主要目的是避免命名空间被污染。[推荐]

```
class SampleStyle
{
public:
    enum Style { Style_Sunken, Style_Thick, Style_Plain };

    void setStyle(Style style);
    void Style style();
};
```

9.1.4 `class` 内的排列顺序

类定义中，请遵循以下排列顺序：

- public 区域在前，protected 区域其次，private 区域最后。[必须]
- 每个区域内同类型成员需放在一起。[必须]
- 每个区域内不同类型成员按照下面的顺序排列：[推荐]
  - 常量定义
  - 自定义类型（`enum/union/struct/class` 等）
  - 静态成员变量
  - 成员变量
  - 构造函数/析构函数
  - 操作符重载函数
  - 多态函数

- 专有函数

9.2 继承与多态

9.2.1 避免多继承

- 多继承很容易带来混淆，而且会降低执行效率，因此要尽可能的避免使用。[推荐]
- 如果实在需要使用多继承，可以通过使用接口继承来代替。[推荐]

9.2.2 不要把不必要的方法放到抽象层次很高的基类中

要保证每个虚函数都是有意义的，不要把不必要的方法放到抽象层次很高的基类中，这会带来以下几方面的问题：[推荐]

- 会增加基类虚表的大小，造成空间浪费
- 影响代码的可读性

9.3 构造函数

9.3.1 避免默认复制构造函数和=操作符

在定义 class 时避免默认复制构造函数和=操作符。[必须]

允许默认复制构造函数是一种很危险的行为。看下面的例子，程序会崩溃！

```
class SampleA
{
public;
    SampleA() { m_ptr = malloc(100); }

    ~SampleA()
    {
        free(m_ptr);
        m_ptr = NULL;
    }
```

```
private:
    void *m_ptr;
};

int main()
{
    SampleA a;
    SampleA b(a);
    return 0;
} // 会导致 m_ptr 二次释放，程序崩溃！
```

另外，复制构造函数可能还会导致 Slicing 的问题。 ※细节请参考其他资料，如侯俊杰的《深入浅出 MFC 2》。

避免默认复制构造函数的原则：

- 如果类在设计的时候就是禁止拷贝的，声明复制构造函数和重载=操作符，但是别写实现
- 如果类在设计的时候是允许拷贝的，而且类中会动态的分配资源，实现复制构造函数并重载=操作符

9.3.2 单个参数的类构造函数使用 explicit 关键字

对单个参数的类构造函数使用 explicit 关键字。[推荐]

Use the C++ keyword explicit for constructors with one argument.

由于单参数的类构造会带有隐式转换的功能，经常导致一些意料外的处理。所以对于单参数的类构造函数必须加上 explicit，如果需要某些类型转换的场合，显式定义转换时的行为。

Normally, if a constructor takes one argument, it can be used as a conversion. For instance, if you define Foo::Foo(string name) and then pass a string to a function that expects a Foo, the constructor will be called to convert the string into a Foo and will pass the Foo to your function for you. This can be convenient but is also a source of trouble when things get

converted and new objects created without you meaning them to. Declaring a constructor explicit prevents it from being invoked implicitly as a conversion.

```
class MyClass
{
public:
    MyClass();
    explicit MyClass(int var); // 使用explicit
    ~MyClass() {}
}
```

10. 函数

10.1 函数参数

10.1.1 避免过多的参数

在代码中，要避免定义带有过多参数的函数。[推荐]

- 影响代码可读性和编码效率

参数过多过长，会造成函数定义很长，甚至需要换行，看起来很不方便。

- 降低编码效率

如果参数过多，则很难被记忆。在写函数调用的时候，要经常在函数定义和代码间切换，一不小心就容易把参数写错。

- 造成性能损失

在函数调用时，参数会压栈；函数调用结束以后，参数还会出栈。如果参数过多，在压栈和出栈的时候就会浪费大量的时间。

- 增大栈内存的使用量

在函数调用时，参数会压栈。如果参数过多，会造成栈内存使用过大。在很多系统中，都会对线程的栈大小做限制，这有可能会导致栈溢出。

对于参数过多的函数，可以通过以下方式进行优化：[推荐]

- 看哪些参数其实是可以避免的。比如有的参数可以通过另外一个参数或者通过其他的方式取得，那么这个参数就是可以避免的。
- 把联系比较紧密的参数打包成结构体，然后通过引用或者指针作为参数传递。
- 如果有些参数在很多函数中都要用，而且使用频率很高，那么不妨把他放到类定义中，作为类成员函数。

10.1.2 避免值传递

函数调用中的值传递很容易带来运行速度问题，特别是对于结构比较复杂的类。值传递的参数，在函数调用过程中会被复制构造，因此造成性能损失。[推荐]

避免值传递的最佳方案是使用引用。[推荐]

10.1.3 注意参数顺序

不合理的参数顺序，容易造成代码中的书写错误，同时也会造成代码阅读困难。

用来共同表示一个内容的参数，请放在一起。[推荐]

比如说，要定义一个 Blit 函数，需要传入源设备和目标设备的信息。其中表示设备信息的有设备 Handle 和几何信息。应该这样写：

```
// 清晰的写法
void Blit(Painter src, Rect srcRect, Painter dst, Rect dstRect);


// 混乱的写法：
void Blit(Painter src, Painter dst, Rect srcRect, Rect dstRect);
```

按照重要性排列参数。[推荐]

比如说，如果函数参数里要传递一个 Buffer 信息，这信息包括 Buffer 的地址和长度。那么应该把地址放在前面，长度放在后面。

```
// 清晰的写法
void InitBuffer(void* buffer, int len);


// 混乱的写法：
```

```
void InitBuffer(int len, void* buffer);
```

参数的顺序要符合自然的思考和语言习惯。[推荐]

比如说，描述一个物体的尺寸，都是先说宽度，后说高度，那么当宽度和高度同时出现在参数中的时候，就应该把宽度放在前面；描述一个人的时候，总是先说身高，后说体重，那么当身高和体重同时出现在参数中时，身高应该放在前面。

10.1.4 形参名不能省略

函数的形参名必须要写，不能省略。[必须]

```
int method(int);          // 不可以！不能省略形参名
int method(int param);    // OK
```

10.2 使用内联函数

积极使用内联函数。[推荐]

- 内联函数能够显著的提高程序运行效率，特别是对于体积短小，调用频繁的函数。
- 在 C++中，最常见的办法就是把函数实现直接写在头文件中。
- 需要注意的是，虚函数是无法内联的，所以虚函数的实现最好还是放在 cpp 中。

一个例子：

```
class Smaple
{
public:
    Sample();

    inline virtual int width() // 不好的做法！虚函数无法内联。
    {
        return m_width;
    }
```

```
privite:
    int m_width;
}
```

10.3 积极使用 const

在函数定义中使用 const 能够很大的提高程序的安全性，但是这是一个经常会被人忽视的领域。建议大家都养成良好的习惯，积极的使用 const。[推荐]

- 如果参数是不需要修改的，请用 const 修饰参数
- 如果不希望在函数调用中修改类成员（比如一些 get 类的函数），请用 const 修饰函数

一个例子：

```
class Smaple
{
public:
    Sample();

    int width() const;                  // 用const 修饰函数
    void setWidth(const int width);     // 用const 修饰参数
privite:
    int m_width;
}
```

11. 内存

11.1 内存分配

- 内存分配以后，一定要判断是否成功。[必须]
  - 在嵌入式环境下，一定不能假设内存分配都是成功的，必要的检查和错误处理是一定要有的。
- 内存分配以后，要先初始化再使用。[推荐]

- 从堆中分配的内存是有记忆效果的，记忆上次被使用后的内容，内存分配函数是不会自动把新分配的内存清 0 的，这个初始化过程必须程序员自己来做。

一个例子：

```
int main()
{
    // 指针初始化为NULL;
    char *name = NULL;

    name = new char[MAX_NAME_LEN];
    // 使用NULL 判断指针是否有效（内存是否分配成功）
    if (NULL != name) {
        // 初始化
        memeset(name, 0, MAX_NAME_LEN);

        ...

        delete[] name;
        // 内存被释放以后，把指针重新设置为NULL
        name = NULL;
    }
    return 0;
}
```

11.2 始终用 NULL 表示无效内存

在任何情况下，都应该用 NULL 表示空指针：[必须]

- 建立空的指针变量的时候，请初始化为 NULL
- 请用 NULL 判断指针是否有效
- 指针指向的内存被释放以后，应该重新设置为 NULL

参考 11.1 中的例子。

虽然在大部分系统上，C 库会把 NULL 定义为 0，但理论上来说 C 库可以把 NULL 定义为任意值，所以始终使用 NULL 而不是 0，才是避免这种情况的唯一方法。

11.3 慎重对待静态内存分配和栈内存分配

11.3.1 内存冗余

动态内存分配的优势之一就是按需分配，用多少分多少，不过静态分配和栈中分配做不到（C99 可以，不过大部分编译器还不支持），这就会造成内容冗余，静态分配导致的内存冗余尤为严重。

11.3.2 静态分配

使用静态分配时，需要考虑到下面因素。[推荐]

- 生命周期无法管理
  - 静态分配的内存，在程序加载的时候就会占用内存，直到程序退出才会被最终释放，这之间会常驻内存。而且由于静态分配内存的冗余性，这对系统资源造成很大的浪费。一般来说，内存一般都不是一直要被用到的，在不需要的时候，静态分配的内存是无法被主动释放的！
- 会造成 ROM 过大
  - 如果在分配静态内存的时候做了初始化，那么这块内存也会同时占用 ROM 空间。大量的静态内存分配很可能会导致 ROM 空间过大。

11.3.3 栈内分配

栈内分配容易导致栈溢出。嵌入式环境下栈一般都很小，在栈中分配内存，特别是有冗余的大内存，很容易导致栈溢出，这个 Bug 是很难被追踪的，所以不要使用。[推荐]

11.3.4 禁止使用可变长的数组定义

虽然 C99 中开始支持可变长的数组定义，即用变量来声明数组的长度。但是由于分配基本是在栈中，如果不对变量大小进行判断，很容易导致栈溢出的问题。所以

- 开发中禁止使用可变长的数组定义。[必须]

遇到此类需求时，推荐改用 new 等堆分配方式。

```
int length = getLengthFromSomething();
int array[length];    // 错误！禁用可变长数组
```

11.4 慎用 C 类型的内存操作 API

11.4.1 不要对非原生类型的变量使用内存操作

`memset`，`memcpy` 对原生类型的变量操作很好，在 C 编程中，处处可见。但是如果用在 C++中，经常会破坏掉虚指针，一定不能对类使用。[必须]

以下是一个破坏虚指针的例子：

```
class Sample
{
public:
    virtual void fn() {}

private:
    int m_data;
};

void MyMethod()
{
    A a;
    memset(&a, 0, sizeof(a));// 错误！会破坏虚函数指针表
    a.fn();// 程序崩溃！因为虚函数指针表被破坏了
    ...
}
```

11.4.2 注意 `memcpy` 的 overlap 问题

`memcpy` 不是一直都正确的，如果目标地址在源地址之后，且地址间有重叠，会出现错误结果：

```
int a[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};


// 期待变为{0, 0, 1, 2, 3, 4, 5, 6, 7, 8}
memcpy(a, &a[1], 9);
// 拷贝以后实际变为{0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
```

C 语言提供的 `memmove` 的函数可以解决此问题，不过带来了性能上的损失。最好的办法不是把所有的 `memcpy` 都改为 `memmove`，而是尽量避免 overlap 的出现；在出现 overlap 的地方，再去使用 `memmove`。`[推荐]`

11.4.3 禁用 `strcpy`, `strcat`, `sprintf`

由于 `strcpy`，`strcat`，`sprintf` 等函数容易导致内存越界的问题，在开发中禁止使用。改用有越界保护的相应函数 `strncpy`, `strncat`, `snprintf`。`[必须]`

12. 注释

通常注释分为两大类：

- 文档注释
  - 为了生成 Doxygen 文档的注释，使用`/** ... */`或者`/// ...`的形式。
- 代码注释
  - 一般的 C++注释，使用`/* ... */`或者`// ...`的形式。

分类	写法	用途
文档注释	`/** 注释 */`	区块说明
	`/// 注释`	区块的简易说明
	`/**< 注释 */`	代码行说明

分类	写法	用途
	`///< 注释`	代码行说明
代码注释	`/* 注释 */`	长文/多行注释
	`// 注释`	简短注释

※注意：注释的内容请使用英文书写。`[必须]`

12.1 文档注释

※注意：下面写的示例中，doxyen 的注释风格只是推荐的写法，实际代码/文件中不做限制。

12.1.1 文件注释

在所有的文件（不仅限于 C/C++的头文件和源文件）开头都需要加入如下的版权信息。`[必须]`

```
Copyright @ 2013 - 2014 Suntec Software(Shanghai) Co., Ltd.
All Rights Reserved.

Redistribution and use in source and binary forms, with or without
modification, are NOT permitted except as agreed by
Suntec Software(Shanghai) Co., Ltd.

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
```

说明：`@2013 - 2014` 中的 `2013` 和 `2014` 可以根据需要修改成合适的年份，例如`@2011 - 2015`。（一般来说，后面一个是当前的年份）

除了版权信息之外，C/C++文件的开始、结尾应该还有下面的注释。

12.1.1.1 头文件注释

- 头文件开始的注释[必须]
  - 版权信息
  - 文件说明
  - 头文件保护宏
  - C++头文件声明（如果是 C++头文件时）

下面示例的内容是 C/C++头文件都必需的。

```
/**
 * Copyright @ 2013 - 2014 Suntec Software(Shanghai) Co., Ltd.
 * All Rights Reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are NOT permitted except as agreed by
 * Suntec Software(Shanghai) Co., Ltd.
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 */
/**
 * @file Sample.h                                          <-- doxygen 的 file 关键字
 * @brief Declaration file of class Sample.                <-- 文件概要说明
 *
 * This file includes the declaration of class Sample, and  <-- 文件详细说明
 * the definitions of the macros, struct, enum and so on.
 *
```

```
 * @attention used for C++ only.
 */


#ifndef SAMPLE_H          <-- 头文件包含保护
#define SAMPLE_H
```

如果是 C++头文件，在头文件包含保护红之后，必须追加下面的内容来注明是 C++头文件。[必须]

```
#ifndef __cplusplus#    error ERROR: This file requires C++ compilation (use a .cpp suffix)
#endif
```

- 头文件结尾的注释[必须]

下面示例的内容是 C/C++头文件都必需的。

```
#endif /* SAMPLE_H */      <-- 头文件包含保护/* EOF */
```

12.1.1.2 源文件注释

- 源文件开始的注释[必须]
  - 版权信息

下面示例的内容是 C/C++源文件都必需的。

```
/**
 * Copyright @ 2013 - 2014 Suntec Software(Shanghai) Co., Ltd.
 * All Rights Reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are NOT permitted except as agreed by
 * Suntec Software(Shanghai) Co., Ltd.
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
```

```
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 */
```

- 源文件结尾的注释[必须]

下面示例的内容是 C/C++源文件都必需的。

```
/* EOF */
```

12.1.2 class 注释

class 声明必须有下面的注释说明。[必须]

要求	项目	关键字	内容
[必须]	概要说明	无	必须写在一行之内
[必须]	详细说明	无	描述详细说明。 如有必要，可以列一些条目形式的内容，或一些 SampleCode
	注意事项	@attention	描述使用 class 时的注意事项。 （例如限定只有某个模块可以使用等）
	警告	@warning	描述警告信息。 （例如：计划删除等等）
	参考	@see	描述一些参考信息。 （例如：应该要看哪些式样书、资料等）
	其他项目	@bug	描述一些还未修改正的已知的 bug 等

写法示例：[推荐]

```
/**
 * 简单的Class 说明。            <-- Class 概要说明
 *
 * 空一行后开始写Class 详细说明。          <-- Class 详细说明
 * 注意这里即使有换行，生成的文档中也不会换行，如果想要换行像下面一样加入空行，
 * 或者使用html 的标签<br>。
 *
 * 这样空一行后的内容会在文档中另起一行，用下面的条目也会单独一行显示。
 * - 条目1
 * - 条目2
 */
class Sample
{
    ...
};
```

12.1.3 函数注释

函数声明必须有下面的注释说明。[必须]

要求	项目	关键字	内容
[必须]	概要说明	无	必须写在一行之内
[必须]	详细说明	无	描述详细说明。 如有必要，如有必要，可以列一些条目形式的内容，或一些 SampleCode
[必须]	参数	@param	先写参数名，留一个空格后描述参数说明。 需要明确输入/输出信息。

要求	项目	关键字	内容
			输入：[IN] 输出：[OUT] 输入输出：[IN/OUT] 没有参数的时候，参数名位置写 `None` 参数有取值范围时，需要说明范围，如有单位也一起注明
[必须]	返回值	@return	描述返回值的意义 如果没有返回值，写成 `None`
	返回值	@retval	有异常返回，或返回值有取值范围时，说明不同取值代表的意义。 一行列举一个（或者一个范围）返回值及其代表的意义。 说明数值范围时，需要注明范围，如有单位也一起注明。 必要时可以写多个`@retval ...` 如果`@return ...`已经可以充分说明时，可以省略 `retval`。
	注意事项	@attention	描述使用 class 时的注意事项。 （例如限定只有某个模块可以使用等） 一般来说，至少要注明是同步/异步接口。 例如， `@attention Synchronous I/F.`

要求	项目	关键字	内容
			或 `@attention Asynchronous I/F.`
	警告	@warning	描述警告信息。 （例如：计划删除等等）
	参考	@see	描述一些参考信息。 （例如：应该要看哪些式样书、资料等）
	其他项目	@bug	描述一些还未修改正的已知的 bug 等

写法示例：`[推荐]`

```
/**
 * 函数概要说明。
 *
 * 函数详细说明。写法和Class 详细说明相同。
 *
 * @param [IN] path  : 年。公历。（范围：1~9999、单位：年）
 * @param [IN] month : 月。（范围：1~12、单位：月）
 * @param [IN] day   : 日。（范围：1~31、单位：日）
 *
 * @return 返回输入日期参数对应的星期几。
 * @retval 0...6:  星期天...星期六
 * @retval 负值  : 输入参数错误
 *
 * @attention Synchronous I/F.
```

```
    */
int8_t getDayOfWeek(int8_t year, int8_t month, int8_t day);
```

函数的概要说明与详细说明之间必须空一行。[必须]

如果函数非常简单，也可以使用简单的注释说明。[推荐]

```
/// 非常简单的成员函数说明。
void simpleMethod();
```

OverLoad 的方法使用`//{@ ... //@}`的方式来说明。[推荐]

此时，必需对所有函数的每个参数依次逐个说明。[必须]

例如，

```
//@{ /**
    * 共同的简单说明。
    *
    * @param [IN] name : 指定名称
    * @param [IN] name : 指定名称
    */
void setName(const std::string name);
void setName(const char* name);
//@}
```

在源文件中，函数注释只需要概要说明。[推荐]

例如，

```
// 函数概要说明
int8_t getDayOfWeek(
                    int8_t year,
                    int8_t month,
                    int8_t day
                    )
```

```
{
    ...
}
```

12.1.4 枚举类型注释

`enum`声明及其枚举值必须有注释说明。`[必须]`

写法示例：`[推荐]`

```
/// 月的枚举定义 enum Month
{
    JAN = 1,    ///<  1 月
    FEB,        ///<  2 月
    ...
    DEC         ///< 12 月
};
```

12.1.5 `struct`注释

`struct`声明及其成员必须有注释说明。`[必须]`

写法示例：`[推荐]`

```
/// MyStruct 概要说明
struct MyStruct
{
    char*   name;   ///< 变量 name 的说明
    int32_t length; ///< 变量 length 的说明
};
```

12.1.6 其他注释

常量以及变量定义，定义前写注释说明。`[推荐]`

```
/// 常量 MAX_PATH_LEN 的说明
```

```
const int32_t MAX_PATH_LEN = 256;


/// 变量name的说明
std::string name;
```

连续定义变量时，也可以把注释说明写在定义后面。[推荐]

```
char*    m_name;    ///< 变量m_name的说明
int32_t m_length; ///< 变量m_length的说明
```

如，class的成员变量推荐使用下面的注释说明的书写方式。[推荐]

```
class MyClass
{
    ...

private:
    int8_t m_year;    ///< 年
    int8_t m_month;   ///< 月
    int8_t m_day;     ///< 日
    ...
};
```

12.2 代码注释的写法

12.2.1 代码注释

代码注释是在说明代码行用途、说明以及注意事项等时适用。

注释的方式有很多种，不过注意不要和 doxygen 预定的注释方式冲突。[推荐]

```
/*****************************************
   不要使用这样的注释！Doxygen会使用类似的注释。
*****************************************/
```

```
//////////////////////
/// 也不要这样进行注释 ///
//////////////////////
```

下面是推荐的几种注释方式。[推荐]

```
// 用于单行注释
/*
 * 用于多行注释
 */
```

12.2.2 不要写无用的注释

注释的目的是增强代码的可读性。如果对可读性没有帮助，那就不要写。[推荐]

好的代码（比如完全遵守本规范的代码），如果能做到命名准确而又有意义，代码行清晰，代码结构明了，那代码的本身就是最好的注释。但是如果代码中涉及到某些特殊的信息，比如为了处理某种特殊情况而打了一个补丁，又或者写了一个很精良但是复杂的算法，那么最好加上注释。

12.2.3 }和#endif后的注释

控制逻辑比较复杂的代码里，经常会出现大量的}和#endif，弄清他所对应的代码经常是要做，但又很麻烦的事情，这种地方请一定要加注释。[推荐]

-

}后面加注释的例子：

-

```
if (condition1) {
    ...
    if (another-condition) {
        ...
    } // End of another-condition
```

```
} // End of condition1
else if (condition2) {
    ...
} // End of condition2
else {
    ...
}
```

-

- `#endif`后面加注释的例子：

-

```
#ifdef COMPILE_OPTION1
    ...
#else // End of COMPILE_OPTION1
    ...
#endif // End of !COMPILE_OPTION1
```

-

12.2.4 用`#if 0 ... #endif`取代大段的代码注释

经常会需要把大段代码注释掉，这种工作下最好是用`#if 0 ... #endif`来做。`[推荐]`

例如，

```
#if 0  // 以下是被注释的代码

    a-large-code-section

#endif // 以上是被注释的代码
```