文档编号:

Infrared Systems

代码规范和常见问题检查表

上海热像机电科技股份有限公司

文档说明

日期	版本	修订者	修订说明	参与评审人员	审核人
2019.12.13	1.0				

目录

一、概述

文档定义通用的代码规范及常见代码问题检查表

二、适用范围

软件研发

三、术语与定义

1. Camel Case

Camel case is the practice of writing phrases such that each word or abbreviation in the middle of the phrase begins with a capital letter, with no intervening spaces or punctuation. Common examples include "iPhone" and "eBay". It is also sometimes used in online usernames such as "johnSmith", and to make multi-word domain names more legible, for example in advertisements.

http://en.wikipedia.org/wiki/Camel_case

2. Pascal Case

首字母为大写的 Camel Case

3. Snake Case / Underscore Case

Snake case (or snake_case) is the practice of writing compound words or phrases in which the elements are separated with one underscore character (_) and no spaces, with each element's initial letter usually lowercased within the compound and the first letter either upper- or lowercase—as in "foo_bar" and "Hello_world". It is commonly used in computer code for variable names, and function names, and sometimes computer filenames

https://en.wikipedia.org/wiki/Snake_case

在本规范中，Snake Case 指所有字母使用全部小写或全部大写，即小写 Snake Case，和大写 Snake Case

四、类 C 代码规范

1. Quick Start

- C/C++

```
typedef std::vector<int> IntVector;    /** Pascal case **/

float g_everyone_may_access;   /** start with 'g_' **/

class ConventionExample        /** Pascal case **/
{
   enum InternalEnum
   {
      IN_VAL1,
      IN_VAL2,
      IN_VAL3,
   };
public:
   void memberFunction(int input_1,int input_2);    /** Camel case **/

   int inlineStyle(int a) { return _private_member; }

   int inlineLongStyle(int a)
   { if(a > 1) return a + 2; else return a; }

   static long static_member_variable;  /** snake case **/
   static const char STATIC_MEMBER_CONSTANT;
private:
   int _private_member;    /** start with '_' **/
};

void              /** return type is highlighted **/
ConventionExample::membmerFunction(int input_1,int input_2)
{
   if(input_1 > input_2)
   {
      ///DO SOMETHING
   }
   else
   {
      ///DO SOMETHING
```

```
    }
}

float
thisIsAnotherFunction(float a)
{
}
```

- Java

```
package com.falcon.activity.main

/** The prefix of a unique package name is always written in all-lowercase
ASCII letters and should be one of the top-level domain names, currently
com, edu, gov, mil, net, org, or one of the English two-letter codes
identifying countries as specified in ISO Standard 3166, 1981.
 **/

public class MainClass{ /** Pascal case **/

    public static final int MIN_WIDTH=4; /** snake case **/

    int age; /** Camel case **/

    void testMain(int age){ /** Camel case **/
        ///DO SOMETHING
    }
}
```

- C#

```
public enum HorizontalAlignment  /** Pascal case **/
{
     Left = 0, /** Pascal case **/
     Right = 1,
     Center = 2
}

public delegate void EventHandler(object sender, EventArgs e); /** Pascal
case **/

public class Authorization
{
     private string _Message; /** Pascal case, Start with  '_'  **/
```

```
    public readonly static int[] VERSION_1_0_1_0 = new int[] { 1, 0,
1, 0 }; /** Snake case **/

    public string Message /** Pascal case **/
    {
      get
      {
          return _Message;
      }
    }

    public Authorization(string token) /** Pascal case **/
    {
      //TO SOMETHING
    }
}
```

2. 命名规范 Naming Convention

命名规则是代码规范的基础，可以使用各自代码的可读性更好。

- 不同的命名规则用来标识不同类型的标识符。
- 强类型的语言变量类型不体现在变量名称中。
- 弱类型的语言变量名可以用小写前缀来标识变量类型。

2.1. C/C++

标识符类型	命名规则	示例
文件名	使用 ASCII 码中的小写字母和数字 使用'_'(下划线)分隔单词 不能使用特殊字符 小写 Snake Case	it_is_an_example.h main.cpp
常量 枚举常量	包括宏定义常量和常量类型变量，全大写字母 使用'_'(下划线)分隔单词 Snake Case	
类型名	Pascal Case	typedef int IntType;

模板类型名		class NewObject; template<typename SomeType> class TemplateObject;
枚举	以 Enum 结尾 Pascal Case	enum ThisIsEnum;
局部变量 公有成员变量 公有静态变量	全小写 使用'_'(下划线)分隔单词 Snake Case	int variable_a char some_local_var
私有成员变量 私有静态变量	以一个'_'(下划线)开头 全小写 使用'_'(下划线)分隔单词 Snake Case	int _member_foo float _val
全局变量	以'g_'开头 全小写 使用'_'(下划线)分隔单词 Snake Case	int g_everyone_knows

普通函数
公有成员函数
	Camel Case	char * getYourName()
		class A
		{
		public:
		void doPublicThings()
		}

私有成员函数	以一个'_'(下划线)开头
	Camel Case
		class A
		{
		private:
		void _doPrivateThings();
		}

2.2. Java

https://www.oracle.com/technetwork/java/codeconventions-135099.html
参考 C/C++的前缀规则

2.3. C#

标识符类型	命名规则	示例
文件名	Pascal Case	MainForm.cpp
常量	全大写字母 使用'_'(下划线)分隔单词 Snake Case	const string TEMP_RANGE

枚举常量	Pascal Case	public enum Alignment
		{
		Left = 0,
		Right = 1,
		Center = 2
		}

类型名 模板类型名	Pascal Case	class NewObject; template<typename SomeType> class TemplateObject;
枚举	Pascal Case	public enum NewEnum
局部变量	全小写 使用'_'(下划线)分隔单词 Snake Case	int variable_a char some_local_var
成员变量	以一个'_'(下划线)开头 Pascal Case	int _MemberFoo float _Val
静态变量 属性	Pascal Case	static string Password

成员函数	Pascal Case	string GetYourName()
		class A
		{
		public void DoPublicThings()
		}

2.4. 变量命名原则

- 变量名称体现变量作用
- 越短的名称用在越小的作用域中。

例如：

```
int g_this_is_a_global_variable;

static void
_localJob()
{
   ///some internal job
}

int
thisIsAnInterface(int val_a, int val_b)
```

```
{
    printf("%d %d %d\n",val_a,val_b,g_this_is_a_global_variable);
    for(int i = 0;i < 100;i++)
    {
        printf("%d\n",i);
    }
}
```

- 避免拼写错误（除了有意的单词变形）
- 避免复用变量

同一变量被用于不同的用途不是一个很好的代码风格。

- 单词选择

✓ DO choose easily readable identifier names.

For example, a property named HorizontalAlignment is more English-readable than AlignmentHorizontal.

✓ DO favor readability over brevity.

The property name can_scroll_horizontally is better than scrollable_x (an obscure reference to the X-axis).

X DO NOT use Hungarian notation.

X AVOID using identifiers that conflict with keywords of widely used programming languages.

https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/general-naming-conventions

3. 格式

3.1. 一行不超过 80 个字符

如果一行要写下的内容很多，可以使用换行符，对于字符串，可以使用字符串的连接

```
#define A_VERY_LONG_DEFINE(A,B) if(A + B > 0) \
                                {\
                                    A = B;\
                                }
```

```
char long_string[] = "abcdefghijklmnopqrstuvwxyz"
                     "zyxwvutsrqponmlkjihgfedcba"
                     "abcdefghijklmnopqrstuvwxyz"
                     "~~";
```

3.2. 不同作用域缩进

- 域括号都另起一行，使用代码更松散一些
- 尽量使用空格符进行缩进，不用制表符。可在不同的编辑器中内保持一致的显示。(Makefile 是例外)

```
void
thisIsAFunction()
{
    for(int i = 0;i < 100;i++)
    {
        if(i < 5)
        {
             ///DO SOMETHING
        }
        else
        {
             ///DO SOMETHING
        }

    }
}
```

- Java 域括号的起始括弧不另起一行，结尾括弧另起一行。

3.3. 符号两侧空格的使用

运算符之间一般都需要加一个空格：

```
a = b + c;
```

例外是改变优先级的括号和结尾分号：

```
u = b * (s + t);
```

对于内联函数，大括号内侧也有一个空格：

```
int inlineFunction()
{ return 1000; }
```

3.4. 函数声明和实现

函数声明于一行中，过长的时候，变量类型开始换到下一行：

```
void showYouASimpleFunction();
void thisIsAVeryLongList(int a1,int a2,int b1,int b2,
                         int c1,int c2,int & ret);
```

函数实现时，将返回值单起一行：

```
void
againSimpleFunction()
{
    ///DO SOMETHING
}

std::vector<std::string>
longReturnName(std::string str)
{
   ///DO SOMETHING to r
   return r;
}
```

内联函数可以使用更简洁的单行和双行方式，注意控制每行代码不要过长。

```
...
int inlineStyle(int a) { return _private_member; }

int inlineLongStyle(int a)
{ if(a > 1) return a + 2; else return a; }

...
```

4. 注释的使用

- 注释不是越多越好。
- 准确的命名是一种很好的注释。
- 清晰的代码逻辑是一种很好的注释。

注释使用'///'或'/**'来表示有意义的注释。其它的注释都认为是可以去除的临时注释。

以下列举一些必须使用注释的地方：

4.1. 库接口

所有库接口应当有准确详细的注释，

- 说明接口的用途、每个参数及返回值的意义、以及可能带来的副作用。
- 对于类库应当指出可能抛出的异常
- 对于一些应用，应当指出可重入性和线程安全性

4.1.1. C/C++

```
/** REST DELETE 命令
     DELETE 一般没有正文返回，buf 可以为空(NULL)
     输入：
        c       RestConn 连接句柄
        path    URL 路径
        hdr     额外的 HTTP 头,可以为空(NULL)
        buf     接收到的正文将返回在此缓冲区中
        bufsz   输入的 buf 的大小
     输出：
        length  远端返回的正文长度

     返回：
        网络通信正常时返回 HTTP 状态码，其它情况返回负值
**/
RESTC_API int restc_del(RestConn c,__IN const char * path,RestHeaders
hdr,__OUT char * buf,size_t bufsz,__OUT size_t * length);
```

4.1.2. Java

```
/**
* This method demonstrates square().
* @param args Unused.
* @return Nothing.
* @exception IOException On input error.
* @see IOException
*/
public void test(String args[]) throws IOException {
     // DO SOMETHING
}
```

4.2. 可能会引起误解的代码

```
...
```

```
  a = b + c;
  c = a + b;  /** 变更 a 的赋值会有副作用，a 是某个硬件寄存器**/
...
```

5. 引用其它来源的库代码时

引用的部分保持原有的名称，所有输入参数及返回应当使用本文档的规范。

```
#include <externalLib.h>
#include <our_lib.h>
...

void
ourFunction(int input_1,int input_2)
{
   int our_local = ExternalLib_GetSomething(input_1,input_2);
   ...
}
```

五、其它语言

一些脚本语言本身对代码有格式的要求，应当遵守。

1. Python

https://www.python.org/dev/peps/pep-0008/
标识符的规则应当参考 C/C++定义。

2. Linux C

由于 Linux 代码的特殊性，请按照 Linux 文档代码规范，Documentation/CodingStyle

3. DLL C

Windows DLL 接口定义有其特殊性，为了保证最大程度的兼容不同语言平台，DLL 的

接口只能使用标准 C 来定义。

- 函数标识符命名只使用单一的全小写下划线连接（Snake Case）标识符。
- 库标识符应当清晰并作为所有标识符的前缀，以避免名空间污染

```
/*
 * StreamSDK 1.2
 * Copyright 2017, IRS.cn
 *
 * 获取网络型热像仪流数据的接口库
 * library for grabbing stream data from IIR device
*/

#ifndef STREAMSDK_H_INCLUDED   /** 防止头文件重复包含的可移植方法 **/
#define STREAMSDK_H_INCLUDED

#pragma once

/** 方便移植到非 windows 平台 **/
#ifdef _WIN32
#ifdef STREAMSDK_EXPORTS
#ifdef __cplusplus
#define STREAMSDK_API extern "C" __declspec(dllexport)
#else
#define STREAMSDK_API __declspec(dllexport)
#endif
#else
#ifdef __cplusplus
#define STREAMSDK_API extern "C" __declspec(dllimport)
#else
#define STREAMSDK_API __declspec(dllexport)
#endif
#endif

#ifndef CALLBACK
#define CALLBACK __stdcall
#endif

#else
#ifdef __cplusplus
#define STREAMSDK_API extern "C"
#else
#define STREAMSDK_API
#endif
#ifndef CALLBACK
```

```
#define CALLBACK
#endif
#endif


#ifndef TRUE
#define TRUE 1
#endif

#ifndef FALSE
#define FALSE 0
#endif

/** 标识指针的作用 **/
#define __IN
#define __OUT
#define __INOUT

#pragma pack(push,8) /** 保证结构体的大小一致 **/
enum streamsdk_enum_error_code
{
    STREAMSDK_EC_OK = 0,                /** 函数调用成功 **/
    STREAMSDK_EC_FAIL = 1,              /** 函数调用失败 **/
    STREAMSDK_EC_TIMEOUT = 2,           /** 网络超时 **/
    STREAMSDK_EC_LIMIT,
};

...

/** 帧缓冲区描述 **/
typedef struct streamsdk_st_buffer_
{
    void *          buf_ptr;            /** 帧缓冲区指针 **/
    unsigned int    buf_size;           /** 帧大小 **/
    unsigned int    buf_pts;            /** 帧序列号 **/
} streamsdk_st_buffer;

...

/** 数据帧回调函数 **/
typedef void (CALLBACK * streamsdk_cb_grabber)(int error,__IN streams
dk_st_buffer * ,__IN void * user_data);
#pragma pack(pop)
```

```
STREAMSDK_API int streamsdk_get_version(__OUT int * ver_no);
```

4. JSON

- 键命名使用小写和连字符"-"
- 对象左括号'{'不另起一行，'}'另起一行。
- 冒号':'后带一个空格

```
{
   "capturer": {
      "ip": "192.168.1.97",
      "port": 10080
   },
   "device": {
      "init-vsk": 1500,
       "model": "PICO384-2",
       "vsk": [0, 1023],
       "gfid":[0, 255],
       "ad": [0, 16383],
       "linear-win": 2000
   }
}
```

六、检查表

1. 代码的非功能要求

检查项	说明
可维护性 Maintainability	· 可读性Readability 代码应当具有自解释性。 · 可测试性Testability 代码容易测试，将功能分配到不同的函数中以便分开测试 · 可调试性Debuggability

	日志（记录控制流和参数数据等）、异常信息等 · 可配置性Configurability 可以通过外部参数进行配置
可重用性 Reusability	减少复制粘贴
可靠性 Reliability	异常及资源回收
可扩展性 Extensibility	扩展新功能
安全性 Security	
性能 Performance	
可伸缩性 Scalability	当数据量变大时，是否能快速扩展性能
可用性 Usability	考虑终端用户体验

2. 面向对象分析与设计 OOAD

检查项	说明
单一响应原则 Single Responsibility Principle	不要在一个类或函数中放入多种响应
开关原则 Open Closed Principle	添加新函数时不能修改已有的代码
Liskov substitutablility principle	子类不改变父类的行为，子类可以作为基类的替代
接口隔离 Interface segregation	不要写超级大的接口，将它们按功能分成更小的接口，接口中不应当有与功能无关的参数
Dependency Injection	在代码中加入中间代理层以减少关联性，提升代码的扩展性和通用性。

https://www.evoketechnologies.com/blog/code-review-checklist-perform-effective-code-reviews/

3. Coding best practices

No hard coding, use constants/configuration values.

Group similar values under an enumeration (enum).

Comments – Do not write comments for what you are doing, instead write comments on why you are doing. Specify about any hacks, workaround and temporary fixes. Additionally, mention pending tasks in your to-do comments, which can be tracked easily.

Avoid multiple if/else blocks.

Use framework features, wherever possible instead of writing custom code.

4. Code Review Tips

https://www.evoketechnologies.com/blog/simple-effective-code-review-tips/

Software code review is a process to ensure that the code meets the functional requirements and also helps the developers to adhere to the best coding practices. Additionally, code review process helps in improving the software quality.

Based on my experience, would like to share 10 simple code review tips, which would help code reviewers and software developers during their code reviews.

1. Highlight issues in the code

Never force software developers to change the code written by them. It may hurt their ego and they may repeat the same mistake if they do not understand the reason behind code change recommendation. Highlight the issues in the existing code and its consequences.

Here's an interesting quote on this point: "If an egg is broken by outside force, life ends. If broken by inside force, life begins. Great things always begin from inside." – Jim Kwik, Learning Expert.

2. Explain relevant principles

If software developers hesitate to accept given suggestions/recommendations, then explain them relevant principles such as Separation of Concerns, SRS (Single Responsibility Principle), Open-Closed principle, Cyclomatic complexity. If necessary, discuss with them the Non Functional Requirements (NFR) such as Maintainability, Extensibility, Testability and Reliability.

3. Discuss relevant quotes

To make the code review process more interesting and engrossing, remind developers relevant quotes/proverbs:

"Any stupid can write the program that computer understands but only good programmers write code that humans understand" – Martin Fowler

"Measuring programming progress by lines of code is like measuring aircraft building progress by weight." – Bill Gates

"Fat model and thin controller", "High cohesion and Low coupling"

"When debugging, novices insert corrective code; experts remove defective code." – Richard Pattis

4. Do few things offline

Instead of explaining the entire solution to developers during the code review process, simply share the links of relevant websites or encourage them to research on the internet by providing keywords. This action would certainly save the code reviewer's time and energy. And of course, developers would also like it, since they too need some time to assimilate the proposed solution.

Instead of always sitting next to a developer during the code review process, code reviewers should obtain the code from the source control or shared path, so that it saves developers time. And this would also give code reviewers ample time to recommend the best solution in the context.

5. Consider as an Opportunity to learn best practices

Sometimes software developers may take the code reviewer's comments personally and defend the code without a valid reason. It then becomes the responsibility of a code reviewer to inform the developer to consider this exercise as an opportunity to learn/discuss best practices, but not to identify issues to criticize. Ideally, code reviewers should inform the managers that code review comments should not be used to assess a software developer's skills. Code review should always be done in a competitive spirit to find more useful comments.

6. Always be patient and relook if required

Sometimes, developers do not accept suggestions/recommendation and keep debating. A code reviewer many not know the exact context and challenges, when the code was written. A code reviewer should understand all the points being made by the developer without losing patience. Furthermore, to make the point crystal clear, a code reviewer can explain the points on a paper or on a whiteboard by comparing the developer's approach vs code reviewer's approach. Every approach has its pros and cons, need to choose the right approach, whichever weighs more after careful evaluation.

Many times, a third approach evolves which is acceptable to both the developer and the code reviewer. If both of them do not come to a conclusion, then stop the discussion by saying "Let's discuss this tomorrow, after doing some more analysis". If the same issue is re-looked on another day with a fresh mindset, it is quite likely that a new approach evolves. Always remember that "No problem can be solved from the same level of consciousness that created it." – Albert Einstein

7. Explain the need for best coding practices

Generally, software developers mention that best coding practices are not followed due to tight project schedules. Developers may feel that it is an acceptable practice. However, code reviewers should educate software developers that as the code size increases or after sometime, the application becomes very difficult to maintain. Moreover, if a client verifies the code then poor quality code may give wrong impression on the team's/organization's quality standards. It may also impact awarding new projects or referring an organization to prospective clients.

If the project schedules are too tight then code reviewers should suggest developers to perform code refactoring while fixing a defect/adding an enhancement or in next version. While refactoring the code, some functionality may break accidently. Code reviewers should convince the project managers by explaining the importance of code refactoring

and the need for allocating additional time to plan this activity.

8. Consult second level code reviewer (if not convinced)

If a code reviewer recommends few suggestions, but the developer hesitates to accept these, then discuss it out with the developer. It is quite possible that the code reviewer may not know the entire context. If the developer is still not convinced with the recommendations of the code reviewer, it is perfectly all right to consult a second level code reviewer. However, the developer should ensure that second code reviewer's suggestions are forwarded to the first code reviewer to ensure that everyone is on the same page.

9. Capture the enhancements and technical debt

It is quite likely that some code review suggestions cannot be implemented during current release. However, a code reviewer should ensure that all accepted recommendations are clearly documented in a shared code review document, so that these are implemented at an appropriate time in future. Additionally, code reviewer should identify and capture all the enhancements from technology and business perspective. Once the project is completed, all captured enhancements can be considered for implementation, instead of searching at that moment. Finding enhancements during code reviews is more efficient than finding separately at the end of the project.

10. Document all code review comments

Document all code review comments in an email, word document, excel, or any standard tool used by the organization. Making a mistake for the first time is acceptable, but it is not a good sign to repeat the same mistake. The code review document helps software developers to cross check the highlighted issues and avoid making similar mistakes in future. Additionally, maintaining a code review document is a mandatory part of the Capability Maturity Model Integration (CMMI) level process.

Conclusion

The above code review tips would help code reviewers and developers to perform effective code reviews. The code review process should always be pursued in a constructive way by all stakeholders to gain maximum benefit.

The code review process not only improves the software quality but also helps software developers to enhance their skills continuously. Hence, organizations/project managers must ensure that code reviews are made integral part of software development lifecycle.