Meta Team C++ Style Guide

更新

- 2019.1.12 首次发布

概述

本文档是 Meta 战队 C++ 代码规范，适用于 Meta 战队编写的各种代码。推行统一的代码规范的目的有：

- 减少因不良代码风格而带来的错误和 Bug
- 增加代码可读性与可维护性
- 减少协作成本、重构成本、测试成本
- ~~满足控制组长的强迫症~~

目录

1 命名规范

命名总体原则是清晰、易读。

- 不可使用无意义命名，例如 `xyz`，但一些局部变量可以使用常见命名，例如 for 循环使用的 `i`
- 不可使用拼音
- 不可使用意义不明的缩写，但常见缩写可以使用，例如 `config` 或 `conf` 用于表示 configuration。如需自定义缩写，务必在显眼处书写注释

1.1 文件命名

代码文件应写在 .cpp 文件中，一般而言，每个 cpp 文件应有一个同名 .h 头文件。

文件名应全部小写，可以用下划线（_）作为分隔，例如：

`remote_interpreter.cpp`
`mcuconf.h`

1.2 类命名

类名应每个单词首字母大写，可以用下划线（_）作为分隔，缩写单词则全部大写，例如：

```
class GimbalInterface {}
class LED_Thread {}
```

1.3 类型命名

struct、enum 应以 `小写字母 + 下划线（_）分隔 + “_t“ 结尾` 命名，可使用 `typedef` 为匿名 struct/enum 命名，例如：

```
enum rc_status_t {
    REMOTE_RC_S_UP = 1,
    REMOTE_RC_S_DOWN = 2,
    REMOTE_RC_S_MIDDLE = 3
};

typedef struct {
    float ch0; // normalized: -1.0(leftmost) ~ 1.0(rightmost)
    float ch1; // normalized: -1.0(downmost) ~ 1.0(upmost)
    float ch2; // normalized: -1.0(leftmost) ~ 1.0(rightmost)
    float ch3; // normalized: -1.0(downmost) ~ 1.0(upmost)
    rc_status_t s1;
    rc_status_t s2;
} rc_t;
```

1.4 变量、函数、方法命名

变量、函数、方法使用 `小写字母 + 下划线（_）分隔` 或第一个单词小写，之后单词首字母大写（驼峰命名法），例如：

```
int feedback_raw_angle;
int feedbackRawAngle;

void start_receive();
void startReceive();
```

内部使用的变量、函数、方法，尽可能使用访问控制。对于暴露给外部的变量、函数、方法，可以使用以下划线开头的命名方法，标示内部使用，例如：

```
int _key_code; // hold key code data, only for internal use
```

模块初始化代码可写在类的构造函数中，但需要 HAL 或 ChibiOS 初始化后执行的代码则不可，一般这一部分代码会放置在命名为 `start` 或 `begin` 的函数/方法中。

1.5 常量命名

常量尽可能限制在对应的 Scope 中，命名方法与变量一致，亦可使用以 `k` 开头表示常量的命名方法，例如 `kSerialSpeed`。如需暴露在较大的 Scope 中，需增加特有前缀。

1.6 枚举命名

枚举以 `全大写 + 下划线（_）分隔` 命名，最好带有特定前缀，且限制在相应的命名空间中，例如：

```
class Remote {
public:
    enum rc_status_t {
        REMOTE_RC_S_UP = 1,
        REMOTE_RC_S_DOWN = 2,
        REMOTE_RC_S_MIDDLE = 3
    };
}
```

2 注释与日志规范

注释很重要！注释很重要！注释很重要！写注释的目的是为了便于下一个读代码的人（很有可能是你自己）理解，因此，写注释应从读者角度出发，以便于读者理解为准则。

注释使用 `英文` 书写。

2.1 CLion 中的三种注释格式

在 CLion 中，对以下三种注释有独特的代码高亮。

2.1.1 普通注释

包含 `//` 和 `/**/` 注释，在 CLion 中以灰色显示。

```
// Here is a normal comment
/* Here is another normal comment */
```

代码后同一行的注释，使用 两个空格 + // + 一个空格 + 注释内容 的格式：

```
1 uint16_t last_angle_raw;  // the raw angle of the newest feedback, in [0, 8191]
```

独立成行的注释，使用 // + 一个空格 + 注释内容 的格式。注释内容句首字母大写：

```
1 // Fill the header
2 txmsg.IDE = CAN_IDE_STD;
3 txmsg.SID = 0x1FF;
4 txmsg.RTR = CAN_RTR_DATA;
5 txmsg.DLC = 0x08;
```

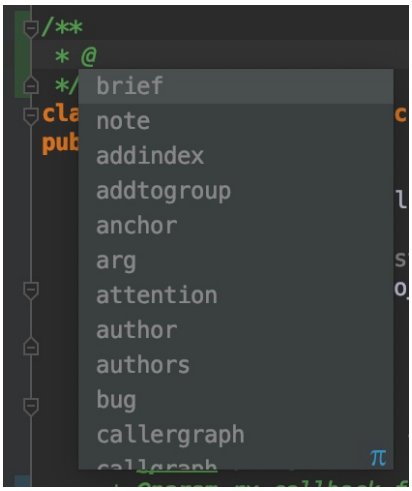
2.1.2 多行高亮注释

以 /** 开头，*/ 结尾的注释在 CLion 中会以绿色高亮显示。

```
/**
 * @brief initialize a can interface
 * @param driver              pointer to can driver
 * @param rx_callback_func    pointer to callback function when a message is received
 */
CANInterface(CANDriver* driver, can_callback_func rx_callback_func) {
    can_driver = driver;
    rx_callback = rx_callback_func;
}
```

```
/** CAN configurations **/
CANConfig can_cfg = {
        CAN_MCR_ABOM | CAN_MCR_AWUM | CAN_MCR_TXFP,
        CAN_BTR_SJW(0) | CAN_BTR_TS2(3) |
        CAN_BTR_TS1(8) | CAN_BTR_BRP(2)
};
```

多行高亮注释可包含以 @ 开头的关键字，CLion 的自动补全会提供常见的关键字，并将关键字加粗加下划线。



常用的有：

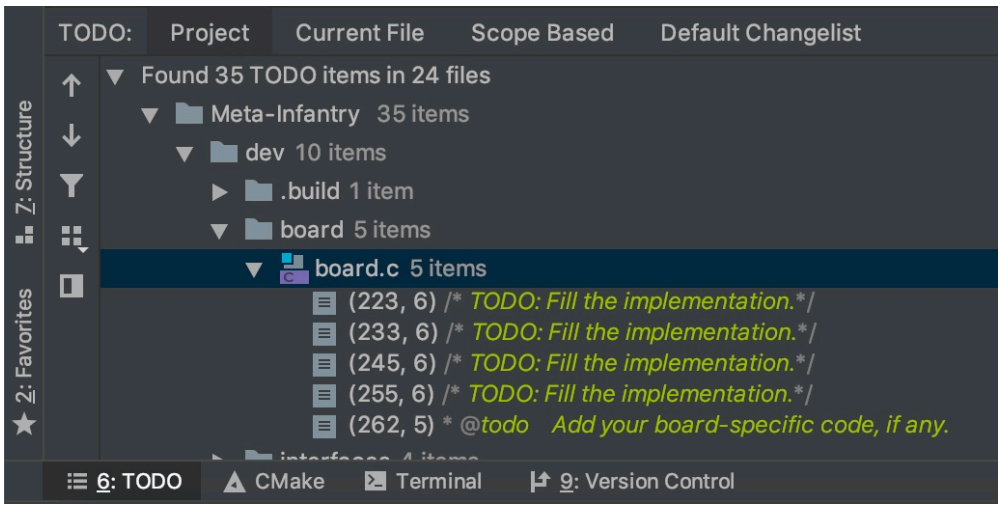
- @brief：概述、简介
- @note：特别说明
- @params：方法/函数参数，见 3.3 方法/函数注释。
- @return：方法/函数返回值，见 3.3 方法/函数注释。

2.1.3 TODO 与 FIXME 注释

```
// TODO: Handle failure of this function

// FIXME: this function crashes when CAN fails to initialized
```

这两种注释会以亮绿色显示，且 CLion 提供了快速查看这两种注释的功能。



2.2 文件注释

每个文件开头应包含作者、文件创建时间、修改记录等信息。如文件受版权保护，则也应包含版权公告。可使用 CLion 自动生成的文件注释。

2.3 类、结构体、枚举注释

类、结构体、枚举前应有注释，描述其用途与注意事项，除非其用途非常显而易见。使用 `多行高亮注释`。

```
/**
 * A serial shell interface.
 * Using SD6 (UART6) as serial port.
 */
class Shell {
```

2.4 方法/函数注释

每一个方法/函数应有对应的注释，用于描述函数的用途、各个参数、返回值、注意事项等。使用 `多行高亮注释`。值得一体的是，将注释在 .h 和 .cpp 文件中复制并没有太大意义。一般而言，建议将最完善的注释放在头文件中。

```c
/**
 * @brief convert string to signed integer
 * @param s         the string to be converted
 * @return the integer
 * @note NO ERROR CHECK
 */
static inline int atoi (const char* s) {
    int ret = 0;
    const char* p = s;
    if (*p == '-') p++;
    while (*p) {
        ret = ret * 10 + (*p - '0');
        p++;
    }
    if (s[0] == '-') ret = -ret;
    return ret;
}
```

在 CLion 中，在函数前输入 `/**`，按一下回车，CLion 会自动生成参数与返回值的关键字。

2.5 变量、常量、宏注释

各个变量、常量、宏应当有对应的注释，用以描述其用途，除非其非常明显。对于一些带单位的变量应当写明单位。根据注释长短使用 `普通注释` 或 `多行高亮注释`。

```c
/**
 * Normalized Angle and Rounds
 *  Using the front angle_raw as reference.
 *  Range: -180.0 (clockwise) to 180.0 (counter-clockwise)
 */
float actual_angle; // the actual angle of the gimbal, compared with the front
```

2.6 实现注释

在实现代码中合适位置写下注释，用于概括代码作用、记录巧妙的实现、注意事项等。但注释内容应避免与代码内容重复，反面例子：

```c
if (failure) return false;  // return false if failed
```

2.7 TODO 与 FIXME 注释

顾名思义，TODO 注释用于记录之后需要增加、改进、修改的事项，FIXME 注释用于记录需要修复的事项。这两种注释不应长久保留在代码中，应及时处理或移除。

2.8 Git 日志

Git commit 时需要详细书写日志，明确说明本次 commit 做的更改，这能帮助其他成员快速了解 commit 的内容，如果之后需要版本回退，有完整的日志也能帮助快速定位回退位置。

提交日志包含以下内容：

- 增加的模块、功能
- 修复的问题
- 目前代码还存在的问题，下一步计划
- ...

日志使用**使动语态**书写：

```
I fix a bug in GimbalController. // Bad
```

```
2 Fix a bug in GimbalController that protection fails when both motors are disabled. // Good
```

3 头文件

3.1 ifdef 保护

每个头文件都应有 ifdef 保护，防止一个头文件被包含多次。宏命名规则为 `工程_文件名_后缀名`，应全部大写，或使用 CLion 新建头文件时自带的 ifdef 保护。例如：

```
1 #ifndef META_INFANTRY_CAN_INTERFACE_H
2 #define META_INFANTRY_CAN_INTERFACE_H
3
4 // Header content
5
6 #endif // META_INFANTRY_CAN_INTERFACE_H
```

3.2 定义书写

各种定义应当写在头文件中，而函数声明、函数或方法定义则应当写在对应的 cpp 文件中。但在类的定义中，较短的函数或方法代码亦可直接写在类定义中。

3.3 注释书写

头文件是明确模块接口的地方，因此头文件中应当**详细书写**接口注释、模块注释等。注释书写规范参考 注释与日志规范 一节。

4 模块化、封装与访问控制

C++ 提供了类、命名空间等语言特性用于封装，访问控制也非常灵活高效，在开发过程中应当注意代码的模块化与封装。

4.1 模块化

模块化的主要目的有：复用代码、便于测试、便于协作。封装主要通过类来实现。

例如使用经过封装的 `GimbalInterface` 模块和 `GimbalController` 模块，主线程代码只需要这样写：

```
1  // Use Remote::rc.ch0 * rc_yaw_angle_angle as target angle
2  // Input target angle and actual angle to get target velocity
3  yaw_target_velocity = GimbalController::yaw.angle_to_v(GimbalInterface::yaw.actual_angle,
4                                                          Remote::rc.ch0 * rc_yaw_angle_angle);
5
6  // Input target velocity and actual velocity to get target current
7  GimbalInterface::yaw.target_current = (int)
   GimbalController::yaw.v_to_i(GimbalInterface::yaw.angular_velocity,
8
   yaw_target_velocity);
9  // Send instructions to motors
10 GimbalInterface::send_gimbal_currents();
```

主线程只需要控制各个模块间的数据流动、调用函数即可，如果需要更改控制逻辑，也只需要修改主线程的代码。

4.2 封装

各模块的独立，主线程代码的简洁、测试的简便均有赖于模块的封装。简而言之，封装即是将内部使用的变量、代码等隐藏，对外只提供必须的接口。

Q：是不是只要封装做得好，内部的代码就可以不遵守规范为所欲为了？（滑稽）
A：（严肃脸）不，你还是会被打

4.2.1 结构体、枚举、常量封装

一个类特有的结构体、枚举、常量应置于类的命名空间下 。正确示例：

```
class MotorController {
public:

    typedef enum {
        YAW_ID,
        PITCH_ID
    } motor_id_t;

    motor_id_t id;
};
```

错误：

```
typedef enum {
    YAW_ID,
    PITCH_ID
} motor_id_t;

class MotorController {
public:
    motor_id_t id;
};
```

4.2.2 实例变量/方法 vs 类变量/函数

当一个类需要产生多个实例，且每个实例需要各自独立的变量和方法时，使用实例变量和方法，例如 PIDController，在底盘各个电机、云台电机上均需要独立的 PID 控制器。

当整个程序只需要一个实例时，使用类变量和类函数。例如 Shell，由于整个程序只需要一个 Shell，该类中均为类变量和类函数。在其他代码中可以直接通过命名空间调用而无需定义实例：

```
Shell::start(HIGHPRIO);
Shell::addCommands(gimbalCotrollerCommands);

yaw_target_angle = Shell::atof(argv[0]);
```

4.2.3 宏的使用

由于 #define 不受命名空间限制，如非必须，使用命名空间内的常量代替宏，例如：

```
#define REMOTE_RX_BUF_SIZE 18

class Remote {
    // Other things
};
```

可修改为：

```
class Remote {

private:
    static const int rx_buf_size = 18;

    // Other things
};
```

4.3 访问控制

类的访问控制根据最小需要授权，可以使用友元（friend）给特定的类、函数授权。

在类定义中，将相同授权的成员放在一起，即不要出现多于一个的 `public`、`protected` 和 `private`。

```
class ClassName {
public:
    // Something
protected:
    // Something
private:
    // Something
}
```

参考资料

- Google C++ Style Guide