# C语言代码规范

| 版本号 | 修订日期 | 修订人 | 修订内容 | 备注 |
| --- | --- | --- | --- | --- |
| V1.0.0 | 2022/12/09 | 王玉豪 | 初版规则制定 | |
| | | | | |

## 一、前言

C语言是我们嵌入式使用最多的一种编程语言。在日常工作中，由于C语言没有强制的编码风格要求，导致每个人的代码风格各异，不利于同事们之间的沟通与移植。

编码规范的好处：

- **促进团队之间的合作**：一些较大的项目，我们可以拆分成多个功能，每个人负责不同的功能，最后再合并。如果没有统一的代码规范，那么每个人的代码必将风格迥异，导致合并代码的时候不仅要去处理程序的问题，还得花大量的时间和精力去理解别人的代码。
- **减少BUG的出现**：规范输入输出的参数，对一些异常的处理规范，这样就会在测试过程中减少一些异常以及低端的代码错误引起的低级bug。
- **降低维护成本**：当项目上线逐渐累积，后期的维护成本也有随之提升。例如：A开发完产品，B维护过程中加了一段代码，之后还有C、D等等，这时候每个人的代码还不规范会导致项目维护成本骤增，出现传说中的"屎山上拉屎"的现在，更严重的需要项目重构等，严重浪费人力资源。
- **利于代码审查**：一个合格的项目在上线前后有专人进行代码审查，防止出现一些低端的逻辑性错误。规范的代码的风格，可以让代码审查效率更高，也可以尽快的发现一些bug。
- **自身的成长**：当自身对自己代码规范性越来越严格的时候，自身的能力也会提升的越快，同时也会减少由于代码不规范所引起的低级bug，可以花更多的时间去找问题解决问题的方案。

代码规范的原则：**安全、可靠、可读、可维护、可测试以及可移植**

**PS：修改外部开源代码、第三方代码时，应该遵守开源代码、第三方代码已有规范，保持风格统一，避免因规则、风格冲突而对第三方代码进行修改**

## 二、代码规范

### （一）通用规范

- 编码格式统一采用UTF-8：更好的适配各种环境
- 换行符号采用Unix 的风格 LF，而不采用windows的风格CRLF
- 代码缩进一律为4个空格符号

### （二）头文件

#### 1. 命名规则

头文件的命令应遵循以下规则：

- 按照unix风格，小写的单词用下划线连接
- 包含模块、子模块、功能信息（如有必要）

- 仅使用通用的缩写，不要随意使用字母缩写，缩写必须能够完全的表达其含义
- 如果外部头文件合内部头文件同名，可以在内部头文件名之前加上“_”以区分
- 禁止同名头文件

## 2. 描述信息

头文件的头部需要加上描述性的注释信息，用于描述该文件的作用、版本以及作者等

**头文件头部注释模板：**

```
/**
* @file xx.h
* @author xxxxxx
* @brief xxxx module is used to xxxx
* @version 0.1
* @date 2022-12-06
*
*/
```

## 3. 头文件宏

为了避免头文件重复include导致冲突，每个头文件必须使用宏来避免冲突，宏的命名方式和文件名保持一致，格式：

- 以双下划线开头、结尾
- 文件名中字母大写
- 所有非字母符号转换成下划线
- 所有的头文件内容都必须放在宏保护区域内

如demo_test.h，转换成宏应如下：

```
#ifndef __DEMO_TEST_H__
#define __DEMO_TEST_H__

#endif
```

## 4. extern "C"的使用

为了保证在c函数在c++环境下正常使用，应在头文件使用 extern “c”宏来保证函数声明的确定性。

```
#ifdef __cplusplus
extern "C" {
#endif

#ifdef __cplusplus
}
#endif /* __cplusplus */
```

## 5. 外部头文件引用

头文件引用，必须放在 extern "C"宏之外。

- 保证所有的引用的头文件的必要性，禁止循环依赖，如a.h依赖b.h，b.h依赖c.h，c.h依赖a.h。
- 外部头文件禁止引用内部头文件
- 头文件引用应保持有序、美观

- 外部头文件引用外部头文件的时候需要注意层次，避免依赖倒置
- 内部头文件只能引用本模块的内部头文件，引用的时候需要使用相对路径

### 6. 头文件函数命名规则

详细参考函数规范：函数

### 7. 头文件存放位置

一般的头文件我们都存放在对应的include文件夹下，如有特殊的头文件：例如只给自己本模块代码使用，外部无需调用的可以存放在src文件夹下，防止外部的调用。

### 8. 内联函数

- 谨慎使用内联函数，仅当测试数据证明内联函数有效的提升了性能之后，才可以使用内联函数
- 内联函数的定义、实现都应当放在头文件中
- 内联函数的长度不应超过10行
- 内联函数规范参照：函数

### 9. 数据类型定义

- 仅对外的数据类型需要放到头文件。
- 模块内部使用的数据类型，放到源文件里去定义。
- 数据类型规范参照：类型

## （三）源文件

### 1. 命名规则

相对来说源文件的命名规则和头文件类似：

- 按照unix风格，小写的单词用下划线连接
- 包含模块、子模块、功能信息（如有必要）
- 仅使用通用的缩写，不要随意使用字母缩写，缩写必须能够完全的表达其含义
- 禁止同名头文件

### 2. 描述信息

袁文件的头部需要加上描述性的注释信息，用于描述该文件的作用、版本以及作者等

**源文件头部注释模板：**

```
/**
* @file xxx.c
* @author xxx
* @brief xxx module is used to xxx
* @version 0.1
* @date 2022-12-07
*
*/

/*************************************************************
************************micro define**********************
*************************************************************/


```

```
/*************************************************************
**********************typedef define**********************
*************************************************************/


/*************************************************************
**********************variable define**********************
*************************************************************/


/*************************************************************
**********************function define**********************
*************************************************************/

```

## 3. 函数

### (1) 命名规范

- 遵守unix风格命名方式，以小写字母和下划线组成函数名；

```
int iot_init_params(IN unsigned char *p_init_params);
```

- 对外接口应包含模块、功能信息，以提高可读性；
- 函数命名采用动宾结构的方式，如set_xxx，get_xxx;

```
void user_uart_init(void);
void user_uart_get_data(OUT unsigned char *p_data, OUT unsigned int *p_data_len);
```

- 使用通用的字母缩写，不应该随意使用缩写，以提高可读性；

```
// bad:
VOID sf_register_mqc_cb(VOID)                          // sf，mqc皆为不通用的缩写

// good:
VOID smartframe_register_mqtt_clinet_cb(VOID)          // 如果有明确的通用的缩写，可以使用；没有明确通用的缩写，尽量使用完整单词
```

- 模块内部函数接口应以“__”双下划线开头，并建议以声明为static；

```
// bad:
BOOL_T scan_test_ssid(VOID)

// good:
STATIC BOOL_T __scan_test_ssid(VOID)
```

### (2) 函数的申明

通过头文件引用方式获得函数的声明，而不是使用extern方式；尽量避免使用extern作用与函数。

```
// bad:
extern int mf_init(void);
mf_init();

// good:
#include"mf_test.h"
mf_init();
```

### (3) 函数的形参

函数形参名称应保持在同一行，如需要换行，则需要保持合理的对齐方式

- 参数名应遵守unix风格，以小写字母和下划线，并严格保持一致；
- 使用通用的字母缩写，不应该随意使用缩写，以提高可读性；
- 参数名应该具有明确含义，不要使用无意义的如param1，param2之类的参数；
- 复杂参数应该使用地址传递；
- 参数应该根据其用途进行修饰，如一些入参可以使用const以避免其被修改；
- 每个参数之间，需要保留1个空格作为间隔，保证阅读的流畅性，函数名称和左小括号直接不需要间隔。
- 没有参数的函数应使用void作为参数；

```
// 行宽可以容纳，尽量保持在同一行
void iot_wf_timeout_set(IN int timeout);

// 参数太多、太长，一行无法完全容纳，与第一个参数对齐：
void iot_wf_mcu_dev_init(IN CONST GW_WF_CFG_MTHD_SELcfg,
                         IN CONST GW_WF_START_MODE start_mode,
                         IN CONST IOT_CBS_S *cbs,
                         IN CONST CHAR_T *p_firmware_key,
                         IN CONST CHAR_T *product_key,
                         IN CONST CHAR_T *wf_sw_ver,
                         IN CONST CHAR_T *mcu_sw_ver);

// bad
OPERATE_RET mf_init(IN CONST MF_IMPORT_INTF_S* intf,  IN CONST CHAR_T*
file_name,  IN CONST CHAR_T* file_ver,  IN BOOL_T wrMacEn);
void func();
OPERATE_RET publish_event(CONST CHAR_T* name, VOID* data);//实在没有类型接收可
以使用void *

//good
OPERATE_RET mf_init(IN CONST MF_IMPORT_INTF_S* intf, IN CONST CHAR_T*
file_name, IN CONST CHAR_T* file_ver, IN BOOL_T mac_write_en);
void func(void);
```

**(4) 函数的返回值**

- 函数必须指定返回值；

```
// bad:
func(void);

// good:
void func(void);
```

- 函数内部必须显式return；

```
// bad:
void func(void)
{
    // no return
}

// good:
void func(void)
{
    return;
}
```

- 函数内对于有明确返回值的调用函数的返回值，需要进行判断，并进行异常处理；

```
// bad:
OPERATE_RET foo(void)
{
    BYTE_T cur_channel = 0;
    wf_get_cur_channel(&cur_channel);
    ...
}

// good:
OPERATE_RET foo(void)
{
    OPERATE_RET rt = OPRT_OK;
    BYTE_T cur_channel = 0
    rt = wf_get_cur_channel(&cur_channel);
    if (rt != OPRT_OK) {
        PR_ERROR("wf_get_cur_channel failed!");
        return rt;
    }
    ...
}

```

- 建议内部接口优先使用返回值而不是输出参数，以方便使用者处理，并且更加符合函数的字面上的意义；

```
// bad:
static OPERATE_RET __wf_get_cur_channel(OUT BYTE_T *p_chan);

// good:
static BYTE_T __wf_get_cur_channel(VOID);
```

- 建议外部接口优先使用输出参数而不是返回值，可以保护内部地址信息；

```
// bad:
BYTE_T wf_get_cur_channel(VOID);

// good:
OPERATE_RET wf_get_cur_channel(OUT BYTE_T *p_chan);
```

**(5) 函数的注释**

好的代码应该有自描述性，但是并不是每个coder都可以做到，建议代码应该包含丰富的注释，帮助我们记录、理解、跟踪代码。

- 统一函数的注释风格
- 注释统一使用英文

```
// 函数头部使用多行注释：
/**
* add some comment
*/
void do_something(void);

// 函数内部使用单行注释：
// add some comment
do_something();
intsomething = 0;  // add some comment

// 函数内部注释过长则使用多个单行注释，也可以使用多行注释：
// add some comment1
// add some comment2
do_something();

// 函数内部注释过长则使用多行注释：
/**
* add some comment1
* add some comment2
*/
do_something();

// 注释内容和注释符号需要空一格，注释符号和注释代码最多不要超过4个空格，并列的注释需要保持对齐：
// bad:
//add some comment
do_something();
inta_int_var = 0;//add some comment

// good:
// add some comment
```

```
do_something();
inta_int_var = 0;       // add some comment1
floata_float_var = 0;   // add some comment2

```

- 对外的接口应该在头文件声明，并提供详细注释，详细描述函数功能、参数、返回值

```
/**
* @brief iot_init_params:用于初始iot的指针数据
*
* @param[in] storage_path:
* @param[in] init_params:
* @return OPERATE_RET 0表示成功，非0请参照error code描述文档
*/
OPERATE_RET iot_init_params(IN CONST CHAR_T* fs_storge_path, IN CONST
INIT_PARAMS_S* p_iot_init_param);

```

- 函数头的注释应当具有帮助理解函数用途、参数使用方式，返回值如何处理等，能够帮助使用者了解如何使用该函数，并了解该函数在某些特殊的场景下具有的问题和风险。

```
// bad: 注释无意义，相关信息函数名中已经体现
/**
* @brief iot sdk初始化接口
*
* @param[in] storage_path: 存储路径
* @param[in] init_params: 初始化参数
* @return OPERATE_RET 0表示成功，非0请参照error code描述文档
*/
OPERATE_RET iot_init_params(IN CONST CHAR_T* fs_storge_path, IN CONST
INIT_PARAMS_S* p_iot_init_param);

// good:
/**
* @brief 初始化iot的基本功能组件，分配必要的资源，必须在使用iot功能、接口之前调用以
确保各个功能模块准备就绪
*
* @param[in] storage_path: iot相关的数据库存储的路径，如果不提供则在默认的路径保
存，linux环境想默认为当前可执行程序路径，RTOS环境下不使用此入参
* @param[in] init_params: iot的初始化配置参数，可以根据配置定义iot的一些内部功能
和初始化过程，如果是malloc的内存，则需要使用者进行释放
* @return OPERATE_RET 0表示成功，非0请参照error code描述文档
*
* @warning: 由于KV database初始化比较耗时，如果对事件敏感则需要将init_db设置成
false，并在本函数执行完成之后再显式的对KV database进行初始化
*/
OPERATE_RET iot_init_params(IN CONST CHAR_T* fs_storge_path, IN CONST
INIT_PARAMS_S* p_iot_init_param);

```

- 对内接口按需提供函数头注释，如果不需要进行注释，也可以不提供注释。
- 接口注释需要和函数声明放在一起，函数实现不需要再进行注释，以提高效率，减少工作量。
- 注释应跟随代码更新而及时的更新，避免注释和代码含义不一致。

- 注释尽量做有意义的注释，不要做无意义的注释。

```
// bad: 代码明确的行为不需要额外的注释解释
// gw_wsm read
op_ret = wd_gw_wsm_read(&(gw_cntl.gw_wsm));
if (OPRT_OK != op_ret) {
    PR_ERR("wsm read err:%d",op_ret);
    goto RESET_FAC;
}

// good：注释帮助理解为了下面要做这个判断
// GWCM_LOW_POWER_AUTOCFG和GWCM_SPCL_AUTOCFG不存在lowpower模式
if(((GWCM_LOW_POWER_AUTOCFG == gw_cntl.mthd) ||(GWCM_SPCL_AUTOCFG ==
gw_cntl.mthd)) && \
    GWNS_LOWPOWER == gw_cntl.gw_wsm.nc_tp) {
    PR_NOTICE("wifi config method not fit");
    goto RESET_FAC;
}
```

- 特殊的处理流程、有问题需要规避处理等，必须要注释清楚，避免其他不知情的人修改导致问题

```
STATI COPERATE_RET__wf_reset(IN CONST WF_RESET_TP_T rst_tp, IN CONST
BOOL_T force_clean,\
                             IN CONST GW_WF_CFG_MTHD_SEL mthd, IN CONST
GW_WF_START_MODE wifi_start_mode, \
                             OUT GW_WORK_STAT_MAG_S* gw_wsm)
{
    // wf reset之前要停止网络状态监控定时任务，否则会导致网络状态变化上报，影响应用
层使用
    cmmod_stop_tm_msg(get_gw_cntl()->nw_stat_moni_tm);
    ...
    ...
    return OPRT_OK;
}
```

- 修改第三方代码，必须提供注释，注明修改的原因和验证的结果。注释需按照以下格式：

```
// <comment start-（作者）, 2022-12-08>
// 注释修改原因，是否验证通过
    .....code....
// <comment end-（作者）, 2022-12-08>
```

**(6) 函数的内容**

- 函数内部的变量定义遵循变量相关的规范：变量
- 函数内部表达式遵循表达式的相关规范：表达式
- 函数内部的排版与格式：采用缩进风格编写，每级缩进为4个空格，并保持一致
- 采用K&R缩进风格

```
// bad: 类型定义、逻辑表达式左大括号另起一行，函数实现左大括号没有另起一行
typedef struct
{
```

```
} wf_info_t;

if (_is_mqtt_online())
{
    ...
}
else
{
    ...
}

void func(void) {
    ...
}

// good: 类型定义左大括号不需要另起一行，函数实现左大括号另起一行
typedef struct {
} wf_info_t;

if (_is_mqtt_online()) {
    ...
} else {
    ...
}

void func(void)
{
    ...
}

```

- 长度不应该超过100行，超过100行要进行拆分，提高代码可读性;
- 逻辑层次不应超过3层，超过3层应封装成函数，或者使用逻辑控制的技巧，降低逻辑层次，提高代码可读性;

```
// bad: 嵌套逻辑太深，影响理解
void func(void) {
    if (a) {
        if (b) {
            if (c) {
                if (d) {
                    if (e) {
                        ...
                    }
                }
            }
        }
    }

// good: 拆分函数、简化逻辑，尽量避免过于复杂的逻辑，造成理解门槛
void func1(void) {
    if (d) {
        if (e) {

```

```
        }
    }
    return;
}

void func(void) {
    if (a) {
        if (b) {
            if (c) {
                func1();
            }
        }
    }
    return;
}

// good:  通过优化逻辑，尽量避免过于复杂的逻辑，造成理解门槛
void func(void) {
    if (!a) {
        return;
    }

    if (!b) {
        return;
    }

    if (c) {
        if (d) {
            if (e) {
                ...
            }
        }
    }

    return;
}

```

- 函数体内部的宏定义进行预编译处理的时候，#ifdef不需要留空格

```
// bad: 宏未项格
{
    ...
    #ifdef(GW_SUPPORT_WIRED_WIFI) && (GW_SUPPORT_WIRED_WIFI == 1)
    hal_net_socket_bind(lan_pro_cntl->upd_fd, IPADDR_ANY);
    #else
    hal_net_socket_bind(lan_pro_cntl->upd_fd, ip.ip);
    #endif
    hal_net_set_boardcast(lan_pro_cntl->upd_fd);
    return;
}

// good
{
    ...
```

```
#ifdef(GW_SUPPORT_WIRED_WIFI) && (GW_SUPPORT_WIRED_WIFI == 1)
    hal_net_socket_bind(lan_pro_cntl->upd_fd, IPADDR_ANY);
#else
    hal_net_socket_bind(lan_pro_cntl->upd_fd, ip.ip);
#endif
    hal_net_set_boardcast(lan_pro_cntl->upd_fd);
    return;
}

```

- 函数内对于错误码应该进行记录、处理；

```
// bad: 未处理函数返回值，这些返回有可能为NULL
    cJSON *js_array = cJSON_CreateArray();
    cJSON *js_config = cJSON_CreateObject();
    cJSON_AddStringToObject(js_config, "domain", domain);
    cJSON_AddItemToArray(js_array, js_config);
    CHAR_T *p_buffer = cJSON_PrintUnformatted(js_array);

// good: 处理了函数的异常返回
    root = cJSON_CreateObject();
    if (NULL == root) {
        return OPRT_CR_CJSON_ERR;
    }

```

- 参数合法性校验，内部函数不需要校验，对外的函数需要校验，内部函数的参数合法性需要我们自己保证，外部的参数合法性我们无法保证，只能通过校验规避问题。
- 函数内容根据逻辑合理增加空行，以提高代码的可读性

```
void func(ty_cJSON* root)
{
    if (NULL == root) {
        return;
    }

    _func(root);
    return;
}

static void _func(ty_cJSON* root)
{
    CHAR_T* p_buffer = cJSON_PrintUnformatted(js_array);
    return;
}

```

- 相似内容应该封装成函数调用，以提高代码重用；
- 有重入需求的函数应避免使用全局、静态资源，如果有必要，则需要使用锁、信号量进行保护；
- 遵循单一职能原则，不要一个函数实现多个功能；
- 合理规划统一的出口

```
err:
    kfree(foo->bar);
    kfree(foo);
    return ret;

//bug 在于某些 goto 语句跳转到此时，foo 仍然是 NULL，修复此 bug 的简单方式就是将一个 label 拆分成两个，err_free_bar: 和 err_free_foo:：
err_free_bar:
    kfree(foo->bar);
err_free_foo:
    kfree(foo);
    return ret;

```

## 4. 类型

### (1) 结构体

- 由小写组名、下划线、数字构成
- 尽量简单，但必须能够充分表达其含义
- 通过 typedef 对结构体、联合体、枚举起别名时，尽量使用匿名类型，名称以“_T”结尾
- 若需要指针自嵌套，可以增加 'tag' 前缀
- 成员命名简单明确，并考虑其上下文进行命名，避免造成含义冗余。
- 函数指针以"_CB"作为后缀

```
// bad，成员命名含义冗余
struct msg_head_t {
    enum msg_type_emsg_type;
    int msg_len;
    char* msg_buf;
};

// good
struct msg_head_t {
    enum msg_type_etype;
    int len;
    char* buf;
};

// good
typedef struct {
    enum msg_type_etype;
    int len;
    char* buf;
} MSG_HEAD_T;

// good
typedef struct tag_list_head_t {
    struct tag_list_head_t* prev;
    struct tag_list_head_t* next;
} LIST_HEAD_T;
```

```
27
```

**(2) 枚举体**

- 名称、成员名称都大写
- 成员命名需要考虑完整场景，进行命名
- 以"_E"作为后缀。

```
// good
typedef enum {
    MSG_STATUS_INIT = 0,
    MSG_STATUS_RUN,
    MSG_STATUS_STOP
} MSG_STATUS_E;
```

## 5. 变量、常量

- 遵循“min-length&&max-information”原则，尽量简单，但是能够描述足够充分的信息，命名应以小写字母、下划线、数字组成。
- 不同层次的变量命名，应该考虑其场景，结合上下文进行命名，比如一个局部变量，可以使用“i”、“tmp”这样的名称，但是全局变量、静态变量就不能使用这样的名称。
- 尽量避免使用全局、静态变量，这些变量会造成访问耦合，访问的时候需要进行保护，避免重入问题。
- 变量定义必须进行初始化。

```
// bad
int i;
do_something(i);

// good
int i = 0;
do_something(i);
```

- 合适的、通用的简写以缩短变量的长度，同时保证命名足够简单。

```
// bad
    // networkconfigtype缩写为nc_tp
    // header缩写为hd

// good
    // temp缩写为tmp
    // message缩写为msg
```

- - **局部变量**:简单的、能够表达含义的缩写，需要的话可以使用下划线连接

```
// good
int i = 0;
int len = 0;
BOOL_T station_ap_en = false;

// bad
GW_WF_CFG_MTHD_SEL mthd = GWCM_OLD; // 缩写不能表明其含义。
BYTE_T gw_sigmesh_err_cnt = 0;      // 太长，结合上下文可以直接命名为 err_cnt
int nCount = 0;                     // 统一不用此类风格
```

- **静态局部变量**:需要以"s_"开头；简单的、能够表达含义的缩写，需要的话可以使用下划线连接。

```
// good
static int s_err_cnt =0;
```

- **全局变量**:需要以"g_"开头；简单的、能够表达含义的缩写，需要的话可以使用下划线连接。需要精确的描述使用场景，帮助理解其用途，作用域越大，描述越要精确

```
// good
TIMER_MGR_S g_mgr={0};
```

- **静态全局变量**:需要以"sg_"开头；简单的、能够表达含义的缩写，需要的话可以使用下划线连接。需要精确的描述使用场景，帮助理解其用途，作用域越大，描述越要精确

```
// good
static int sg_cfg_lp_timeout = 900*1000;            //默认3*60s,修改为15分钟=15*60
```

- **常量**:全大写,简单的、能够表达含义的缩写，需要的话可以使用下划线连接。需要精确的描述使用场景，帮助理解其用途，作用域越大，描述越要精确

```
// good
#define PI (3.14)       // 宏
const CHAR_T *BASE_VERSION "40.00"  // 字符串常量

```

- **宏**:全大写。简单的、能够表达含义的缩写，需要的话可以使用下划线连接。需要精确的描述使用场景，帮助理解其用途，作用域越大，描述越要精确

  宏内定义变量，需要在变量名加后缀“_”，避免影响外部的变量命名。函数式宏有非常多的需要注意的问题

```c
// good
#define PI (3.14)
#define SUM(a, b) ((a) + (b))   // Good.

// good，使用do{...}while(0)
#define CALL_ERR_RETURN(func) \
    do {\
    rt = (func); \
    if (OPRT_OK != (rt)) { \
        TAL_PR_ERR("ret:%d", rt); \
           return (rt); \
        } \
    } while(0)
```

- 变量的使用：
  - 定义指针时候需要保持风格一致：

```c
// good，有空格，靠左或者靠右都可以，且必须保持风格一致
int *p1;    // OK.
int* p2;    // bad，  和上一个指针风格不一致.

// bad，没有空格
int*p3;     // Bad: 两边都没空格
int * p4;   // Bad: 两边都有空格

// bad，容易混淆含义
int* a, b;

// good
int *a = NULL;
int *b = NULL;

```

  - 避免不同类型的赋值，避免不同

```c
// bad
BOOL_T ret = 0;
int cnt = 0;    // 冗余初始化，将会被后面直接覆盖
cnt = get_cnt();
char buf[VERY_BIG_SIZE];
memset(buf, 0, sizeof(buf));    // 清空降低性能

// good
BOOL_T ret = FALSE;
char buf[VERY_BIG_SIZE] ={0};

```

  - 尽量避免使用立即数，对于多处使用的数字，必须定义宏或const 变量，并通过符号命名自注释。仅使用一次的数据

```
// bad
INT_T conn_num = 3;

// good
INT_T conn_num = MAX_NUM_OF_CONNECTION;

```

- 变量赋值的时候，不要写在同一行

```
// bad
INT_T i = 0, j = 0;

// good
INT_T i = 0;
INT_T j = 0;

```

- 指针类型的声明和使用必须遵守以下规范

```
// good, 初始化"*"和变量名称相连，并初始化为NULL
INT_T *p1=NULL;
INT_T *p2=NULL;

// good，使用前判断是否为NULL
if (p1) {
    do_something(p1->something);
}

// 分配内存的时候需要判断是否非空，释放的时候需要设置为NULL
p1 = malloc(100);
if (!p1) {
    return;
}
...
free(p1);
p1 = NULL;

```

## 6. 表达式（if,for,switch等等）

- 统一使用K&R缩进风格
- 统一定义缩进为4个空格长度
- 统一定义条件、循环的层次不得超过3层
- 统一定义空格、括号规则
- 关键字后加空格，左小括号后、右小括号前不加空格
- 左大括号和条件、循环条件一行
- 右大括号独占一行

```
// bad
if(condition) {
    action();
}

// good
if (condition) {
    action();
}

```

**(1) 条件语句**

- 禁止条件执行代码在同一行。

```
// bad
if (condition) {action()};

// good
if (condition) {
    action();
}

```

- 不要省略大括号

```
// bad
if (condition)
    action();

// good
if (condition) {
    action();
}

// good
if (condition) {
    do_this();
} else {
    do_that();
}

```

- 只有一个 else有单行，其他 else 有多行的情况需要统一，全部使用括号

```
// bad
if (condition) {
    do_this();
    do_that();
} else
    otherwise();

// good
if (condition) {
```

```
    do_this();
    do_that();
} else {
    otherwise();
}
```

- 在一个条件、循环中超过一个语句的情况也同样需要使用括号

```
// bad
if (condition)
    if (test)
        do_something();

// good
if(condition) {
    if(test) {
        do_something();
    }
}

```

**(2) 循环语句**

- for循环遵循函数的缩进标准，左大括号不需要紧跟着右边小括号，左大括和关键字同一行
- for循环条件里小括号和表达式不需要空一格，运算符之后空一格，以提高代码的可读性

```
// bad
int i = 0;
for (i = 0;i < MAX;i++){
    do_something();
}

// good
int i = 0;
for (i = 0; i < MAX; i++) {
    do_something();
}

```

- 循环条件内不要定义变量，因为我们的编译工具类型非常多，会出现编译问题

```
// bad
for (int i = 0; i < MAX; i++) {
    do_something();
}

// good
int i;
for (i = 0; i < MAX; i++) {
    do_something();
}

```

- do-while：左边大括号紧跟 do关键字，并且空一格；右边大括号和while关键字一行，并且空一格

```
// good
do {
    do_something();
} while(condition);
```

**(3) switch-case语句**

- case单独占一行，并且让case与switch加一个缩进
- 不要遗漏default
- 每个case的执行体**建议**使用大括号保护，避免变量的作用域超出范围

```
// good
switch (suffix) {
    case'A':
    case'a':
        printf("a");
        break;
    case'B':
    case'b':
        printf("b");
        break;
    /* fall through */
    default:
        break;
}

```

**(4) 运算符**

- 在二元操作符和三元操作符周围添加一个空格

```
// good
c = a + b;
d = c > a ? c : b;

```

- 不要在一元操作符之后添加空格

```
// good
++a;
a++;
a = ~b;
c = !a;
```

- 不要在结构体成员操作符周围添加空格

```
// good
a.num = 0;
b->num = 0
```

# 三、 总结

- 规则并不是完美的，通过禁止某一些“在特定情况下有用”的特性，虽然可能会对代码实现造成影响，但是可以为“大多数程序员得到更多的好处”
- 在团队运作中，如果有人认为某个规则无法遵循，可以共同改进该规则，否则应当严格遵守

PS:在刚开始的执行的时候，会让自己写代码的速度变慢，同时会有变扭感，这都是正常的，但是要坚持不懈的去执行，才能让自己代码更加规范，让自己能有进一步的提升，加油!!!