C 语言编程规范

修订历史

版本	日期	作者	修订描述
V1.0	2024/1/8	赵攀	初版

目录

0. 前言

为提高软件代码质量及生命周期，结合我司代码编程中遇到的问题，并参考了业界编程规范近年来的成果，提炼并编写了本编程规范。旨在指导软件开发人员编写出简洁、可靠、高效、可维护、可移植的代码。

本编程规范设计为三大部分，分别从代码可读性，程序鲁棒性以及系统架构一致性进行规定与约束，适用于公司内使用 C 语言编码的所有软件。

1. 代码可读性

1.1 注释

- 文件头部应进行注释，注释必须列出：版权说明、版本号、生成日期、作者姓名、内容、功能说明、修改日志等；

举例：

```
/*
 * Copyright(c) KunGao Corporation, 20xx
 * All rights reserved.
 *
 * $Revision: 1.0$
 * $Date: 20xx-01-19 17:45:59+0800(Wed, 19 Jan 20xx) $
 *
 * Author  : xxx
 * Purpose : Definition for xxx API
 *
 * Feature : The file includes the following module and sub-modules
 *           (1) xxx
 *           (2) xxx
 *           (3) xxx
 *
 */
```

- 函数定义与声明处注释描述函数功能、输入和输出参数、函数返回值等；

举例：

```
/**
 * @brief The API is used for ...
 *
 * @param[in] xxx
 * @param[out] xxx
 * @return xxx
 */
```

- 全局变量注释描述其功能，使用注意事项等；
- 注释应放在其代码上方相邻位置或右方，不可放在下面。如放于上方则需与其上面的代码用空行隔开，且与下方代码缩进相同；

举例：

```
  #endif
#endif

    /* Process input options */
    if (argc > 1)
    {
        ...
    }
```

- 禁止在”//”注释中进行行拼接；

举例（导致注释行下一语句变成注释）：

```
(pb->fpab.free_count)--;     // free counter  \
pe_new->fpae.offset_a = block_index;
```

- 修改代码或函数实现时，只进行有意义的必要注释，可读性好的代码不需要注释;
- 避免在注释中使用缩写，除非是业界通用或标准化的缩写；
- 注释使用的语言默认使用英文；
- 同一产品或项目组注释风格需要统一；
- 函数头（API）等注释格式采用工具可识别的格式，便于用户帮助文档的生成；

1.2 命名

- 标识符的命名要有明确含义，同时使用完整的单词或大家基本可以理解的缩写，避免使人产生误解；

常用单词缩写：

```
argument        可缩写为 arg
buffer          可缩写为 buff
clock           可缩写为 clk
command         可缩写为 cmd
compare         可缩写为 cmp
configuration   可缩写为 cfg
device          可缩写为 dev
error           可缩写为 err
hexadecimal     可缩写为 hex
increment       可缩写为 inc
initialize      可缩写为 init
maximum         可缩写为 max
message         可缩写为 msg
minimum         可缩写为 min
parameter       可缩写为 para
previous        可缩写为 prev
register        可缩写为 reg
semaphore       可缩写为 sem
statistic       可缩写为 stat
synchronize     可缩写为 sync
temp            可缩写为 tmp
```

- 文件命名统一采用小写字符；
- 函数命名应以函数功能或要执行的动作命名；

举例：

```
int32 cdl_acl_add_entry(uint32 group_id, cdl_acl_entry_t *acl_entry);
int32 cdl_acl_remove_entry(uint32 entry_id);
```

- 禁止使用单字节命名变量，但允许定义 i、j、k 作为局部循环变量。全局变量应增加“g_”前缀，静态变量应增加“s_”前缀；
- 对于数值或者字符串等常量的定义，统一采用全大写字母，单词之间加下划线”_”的方式命名（枚举同样建议使用此方式定义）；

举例：

```
#define     MAX_TCAM_HIT_NUM    256
```

- 尽量避免名字中出现数字编号，除非逻辑上的确需要编号；
- 用正确的反义词组命名具有互斥意义的变量或相反动作的函数等；

举例：

add/remove	begin/end	create/destroy
insert/delete	first/last	get/release
increment/decrement	put/get	add/delete
lock/unlock	open/close	min/max
old/new	start/stop	next/previous
source/target	show/hide	send/receive
source/destination	copy/paste	up/down
set/get		

- 除了头文件或编译开关等特殊标识定义，宏定义不能使用下划线”_”开头和结尾；
- 防止局部变量与全局变量同名；
- 重构/修改部分代码时，应保持和原有代码的命名风格一致；

1.3 排版与格式

- 程序块采用缩进风格编写，每级缩进为 4 个空格，相对独立的程序块之间必须加空行；
- 一条语句不能过长（建议行宽值 80），如不能拆分需要分行写；

换行时有如下建议：

① 换行时要增加一级缩进，使代码可读性更好；

② 低优先级操作符处划分新行；换行时操作符应该也放下来，放在新行首；

③ 换行时建议一个完整的语句放在一行，不要根据字符数断行；

举例：

```
if ((TRUE == fdb_DB[idx].hw_valid)
    && (gport == fdb_DB[idx].gport)
    && (!memcmp(mac, (fdb_DB[idx].mac), sizeof(mac_addr_t))))
{
    ...
}
```

- 多个短语句（包括赋值语句）不允许写在同一行内，即一行只写一条语句；

举例：

```
int a = 0;
int b = 0;
int c = 0;
int d = 0;
```

- if、for、do、while、case、switch、default 等语句独占一行，并且，if、for、do、while 等语句后的执行语句增加成对的”{}”，添加”{“的位置独立占下一行；

举例：

```
for (idx = 0; idx < FDB_MAX; idx++)
{
    ...
}
```

- 在两个以上的关键字、变量、常量进行对等操作时，它们之间的操作符之前、之后或者前后要加空格；进行非对等操作时，如果是关系密切的立即操作符（如—>）后不应加空格；

① 逗号、分号只在后面加空格；

举例：

```
int count = split(_tcamInfo, ",", _rlts);
for (index = 0; index < count; index++)
```

② 比较操作符，赋值操作符"="、"+="，算术操作符"+"、"%"，逻辑操作符"&&"、"&"，位域操作符"<<"、"^"等双目操作符的前后加空格；

举例：

```
if (a != TRUE)
a = b + c;
a *= 2;
a = b ^ 2;
```

③ "!"、"~"、"++"、"--"、"&"（地址操作符）等单目操作符前后不加空格；

举例：

```
*p = 'a';
flag = !is_empty;
p = &mem;
i++;
```

④ "->"、"."前后不加空格；

举例：

```
p->id =  pid;
```

⑤ if、for、while、switch 等与后面的括号间应加空格，使 if 等关键字更为突出、明显；

举例：

```
if ()
for ()
switch ()
while ()
```

- 注释符（包括"/*"，"//"，"*/"）与注释内容之间要用一个空格进行分隔；

举例：

```
/* search the first 88-bit */
// Do the first 88-bit lookup
```

- 用括号明确表达式的操作顺序，避免过分依赖默认优先级；

举例：

```
(a + b) * 2;
```

- 避免函数过长，建议新增函数不超过一个屏幕，行数区间（非空非注释行）在[0-50]较为合适；
- 不允许直接使用魔鬼数字；

举例（使用魔鬼数字可读性差）：

```
static int32_t s_tcam_entry_counts[7] = {128, 64, 128, 64, 256, 128, 256};
```

2. 程序鲁棒性

2.1 代码质量

- 使用编译器的最高告警级别，理解所有的告警，通过修改代码而不是降低告警级别来消除所有告警；
- 禁止内存操作越界；

避免内存越界措施：

① 数组的大小要考虑最大情况，避免数组分配空间不够；

② 不要将边界不明确的字符串写到固定长度的数组中;

③ 避免使用危险函数 sprintf /vsprintf/strcpy/strcat/gets 操作字符串，使用相对安全的函数 snprintf/strncpy/strncat/fgets 代替；

④ 使用 memcpy/memset 时，一定要确保长度不要越界；

⑤ 字符串考虑最后的'\0'， 确保所有字符串是以'\0'结束；

⑥ 指针加减操作时，考虑指针类型长度；
⑦ 数组下标进行检查；
⑧ 使用 sizeof 或者 strlen 计算结构/字符串长度，避免手工计算；

- 禁止内存泄漏；

避免内存泄漏措施：

① 异常出口处检查内存/定时器/文件句柄/Socket/队列/信号量等资源是否全部释放；
② 使用指针数组时，确保在释放数组时，数组中的每个元素指针已经提前被释放;
③ 避免重复分配内存;
④ 小心使用有 return、break 语句的宏，确保前面资源已经释放；

- 禁止引用已经释放的内存空间；

避免引用已经释放的内存空间措施：

① 内存释放后，把指针置为 NULL；使用内存指针前进行非空判断;
② 耦合度较强的模块互相调用时，一定要仔细考虑其调用关系，防止已经删除的对象被再次使用;

- 严禁使用未经初始化的变量作为右值。 在首次使用前初始化变量，初始化的地方离使用的地方越近越好;
- 对函数的错误返回码要全面处理;
- 可重入函数使用共享变量时，则应通过互斥手段（关中断、信号量）对其加以保护；
- 废弃代码（没有被调用的函数和变量)要及时清除；重复代码尽可能提炼成函数;
- 通讯过程中使用的结构，必须注意字节序；
- 尽量减少没有必要的数据类型默认转换与强制转换，避免数值溢出，符号错误及截断错误；
- 用宏定义表达式时，要使用完备的括号；除非必要，应尽可能使用函数代替宏；
- 编程时，要防止差 1 错误；

① 此类错误一般是由于把“<=”误写成“<”或“>=”误写成“>”等造成的，应对这些操作符进行彻底检查;
② 使用变量时要注意其边界值的情况;

- 所有的 if ... else if 结构应该由 else 子句结束；switch 语句必须有 default 分支;
- 每个 switch 语句都要至少包含两个分支；
- 不要滥用 goto 语句；
- 确保格式字符和参数匹配；

举例（不正确的格式化字符串会导致程序异常终止）：

```
char *error_msg = "Resource not available to user.";
int  error_type = 3;

printf("Error (type %s): %d\n", error_type, error_msg);
```

2.2 程序效率

提高代码效率，必须在保证软件系统的正确性、简洁、可维护性、可靠性的前提下进行；不能一味地追求代码效率，而对软件的正确、简洁、可维护性、可靠性及可测性造成影响。

- 通过对数据结构、程序算法的优化来提高效率；

① 将不变条件的计算移到循环体外；

② 创建资源库，以减少分配对象的开销；

例如，使用线程池机制，避免线程频繁创建、销毁的系统调用；使用内存池，对于频繁申请、释放的小块内存，一次性申请一个大块的内存，当系统申请内存时，从内存池获取小块内存，使用完毕再释放到内存池中，避免内存申请释放的频繁系统调用。

③ 将多次被调用的“小函数”改为 inline 函数或者宏实现；

3. 系统架构一致性

3.1 代码风格

- 产品/项目组内部应保持统一的代码风格，包括统一的命名，注释，排版等风格；

3.2 头文件设计

- 每一个.c 文件应有一个同名.h 文件，用于声明需要对外公开的接口; 并且文件应当职责单一；
- .c/.h 文件禁止包含用不到的头文件；
- 总是编写内部#include 保护符（#define 保护）;
- 头文件中放置接口的声明，不能放置实现；
- 变量定义不应放在头文件中，应放在.c 文件中；
- 内部使用的函数声明不应放在头文件中；
- 禁止头文件循环依赖；
- 只能通过包含头文件的方式使用其他.c 提供的接口，禁止在.c 中通过 extern 的方式使用外部函数接口、变量。
- 保持模块之间的独立性，不允许包含除了公共模块外的其他模块的头文件；如果 a 模块要用到 b 模块中的函数，不应该把 b.h 包含在 a.h 中，而应该在 a.c 中直接包含 b.h；
- 一个模块通常包含多个.c 文件，建议放在同一个目录下，目录名即为模块名。为方便外部使用者，建议每一个模块提供一个.h，文件名为目录名。

3.2 函数设计

- KISS 原则（Keep It Simple and Stupid），一个函数仅完成一件功能，避免函数过长，功能复杂;
- DRY 原则(Don’t Repeat Yourself)，重复代码需求提炼成函数，避免代码冗余；
- 避免函数的代码块嵌套过深，新增函数的代码块嵌套不超过 4 层；
- 对函数参数需要进行合法性检查;
- 设计高扇入，合理扇出（小于 7）的函数;

扇出是指一个函数直接调用（控制）其它函数的数目，而扇入是指有多少上级函数调用它。扇出过大，表明函数过分复杂，需要控制和协调过多的下级函数，通常函数比较合理的扇出是 3~5。

- 函数的参数个数不超过 5 个;

- 在源文件范围内声明和定义的所有函数，除非外部可见，否则应该增加 static 关键字，且内部函数命名以短下划线开头标记_xxx();
- 使用面向接口编程思想，通过 API 访问数据；如果本模块的数据需要对外部模块开放，应提供接口函数来设置、获取，同时注意全局数据的访问互斥;

3.3 数据结构设计

- 结构功能单一， 不要设计面面俱到的数据结构；结构的定义应该可以明确的描述一个对象，而不是一组相关性不强的数据的集合；
- 一个变量只有一个功能，不能把一个变量用作多种用途；

3.4 编译设计

- 代码至始至终只有一份代码，测试版本与发行版本通过编译开关的不同来实现，并且编译开关要规范统一；

3.5 Log 日志设计

- 在同一产品或项目组内，调测打印的日志需要规范统一；

4. 业界编程规范

- google C++编程指南
- 汽车业 C 语言使用规范(MISRA)