华为C语言通用编程规范

V3.1

目录

0 前言

目的

优秀的代码总是**简洁、可维护、可靠、可测试、高效、可移植**的。
编程是一种创造性的工作。本规范用于引导软件开发人员好的编程习惯，编写出风格一致、容易阅读、高质量的代码，从而提升产品竞争力和软件研发效率。

总体原则

清晰第一

清晰性是易于阅读、易于维护、易于重构的程序必需具备的特征。代码首先是给人读的，好的代码应当可以像文章一样发声朗诵出来。

`程序必须为阅读它的人而编写，只是顺便用于机器执行。` -- Harold Abelson 和 Gerald Jay Sussman

`编写程序应该以人为本，计算机第二。` -- Steve McConnell

简洁为美

简洁就是易于理解并且易于实现。代码越长越难以看懂，也就越容易在修改时引入错误。写的代码越多，意味着出错的地方越多，也就意味着代码的可靠性越低。因此，我们提倡大家通过编写简洁明了的代码来提升代码可靠性。

短小的函数总是更简洁、容易阅读的。 **函数短小是好代码的重要特征之一。**

风格一致

相比个人习惯，所有人共享同一种风格带来的好处，远远超出为统一而付出的代价。
在编码规范的指导下，审慎地编排代码以使代码尽可能清晰、风格统一。

约定

`规则`：编程时**必须**遵守的约定(must)

`建议`：编程时**应该**遵守的约定(should)

范围

本规范适用于公司内使用C语言编程的所有代码。

例外

无论是'规则'还是'建议'，都必须理解该条目这么规定的原因，并努力遵守。
但是，有些规则和建议可能会有例外。

在不违背总体原则，经过充分考虑，有充足的理由的前提下，可以适当违背规范中约定。
例外破坏了代码的一致性，请尽量避免。'规则'的例外应该是极少的。

下列情况，应风格一致性原则优先：
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。

1 命名

命名包括文件、函数、变量、类型、宏等命名。

命名被认为是软件开发过程中最困难，也是最重要的事情。
标识符的命名要清晰、明了，有明确含义，符合阅读习惯，容易理解。

统一的命名风格是一致性原则最直接的体现。

总体风格

常见命名风格有：
驼峰风格(CamelCase)
大小写字母混用，单词连在一起，不同单词间通过单词首字母大写来分开。
按连接后的首字母是否大写，又分: **大驼峰(UperCamelCase)**和**小驼峰(lowerCamelCase)**

内核风格(unix_like)
单词全小写，用下划线分割。
如：'test_result'

匈牙利风格
在'大驼峰'的基础上，加上类型或用途前缀
如：'uiSavedCount', 'bTested'

规则1.1 标识符命名使用驼峰风格

不考虑匈牙利命名，在内核风格与驼峰风格之间，我们选择驼峰风格。
选择驼峰的原因，是考虑我司有大量匈牙利命名风格的存量代码，驼峰与其视觉上、书写习惯上最接近。

类型	命名风格
函数，结构体类型，枚举类型，联合体类型	大驼峰
全局变量，局部变量，局部常量，函数参数，宏参数，结构体中字段，联合体中成员	小驼峰
宏，全局常量(const)，枚举值，goto 标签	全大写，下划线分割

反对匈牙利命名的理由：
匈牙利命名法源于微软，然而却被很多人以讹传讹的使用；而现在即使是微软也不再推荐使用匈牙利命名法。
在命名前增加类型或用途说明，是对类型信息的冗余描述。更麻烦的问题是，如果修改了变量的类型定义，那么所有使用该变量的地方都需要修改；一旦修改有遗漏，则有可能误导读者，产生风险。

命名风格例外：
对于更亲和于 Linux 操作系统的软件，标识符命名可以采用内核风格，并保持统一。 如 Linux内核、驱动(包括用户态与内核态)，BSP软件，Linux社区生态系的软件，等等。

规则1.2 禁止使用生僻的单词缩写，不得使用汉语拼音

简短的命名总是方便阅读的，但前提是容易理解。
在命名中，禁止使用生僻的单词缩写，但使用常见、通用的缩写是允许并被推荐的。
某个系统的专用缩写，或局部范围内形成共识的缩写，也是可以的。

一些常见可以缩写的例子：

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

建议1.1 作用域越大，命名应越精确，必要时可以增加模块前缀

C 与 C++ 不同，没有名字空间，没有类，所以全局作用域下的标识符命名要考虑不要冲突。
对于全局函数、全局变量、宏、类型名、枚举名的命名，应当精确描述并全局唯一。

例：

```
int GetCount();                 // Bad: 描述不精确
int GetActiveConnectCount();    // Good
```

模块前缀与命名主体之间，可以按驼峰方式连接，也可以全大写使用下划线(_)连接。
选择一种风格，并保持产品内部统一。
示例：

```
int PREFIX_FuncName();      // OK: 全大写，下划线连接
int PrefixFuncName();       // OK: 驼峰方式，形式上无前缀，内容上有前缀

struct XXX_MyType {         // OK.
    ...
};

enum XxxMyEnum {            // OK.
    ...
};
```

模块前缀应当尽量简短。
模块前缀一般不能超过**2**级，前缀过长会影响命名主体，喧宾夺主。

```
int XXX_YYY_FuncName();     // OK. 两级前缀，大模块加小模块
int XxxYyyFunc();           // OK.

int XXX_YYY_ZZZ_FuncName(); // Not Good: 超过两级前缀则过长。
```

继承既有通用约定接口的函数，命名沿袭原有命名，可以直接在原有名字后面增加短后缀，如安全函数:

```
memcpy_s(...);  // OK: 继承 memcpy 库函数命名，增加后缀 '_s'
```

文件命名

建议1.2 文件命名统一采用小写字符

文件名统一采用全小写字母命名；用下划线(_)分开。禁止使用空格。
文件名应尽量简短、准确、无二义性。
不大小写混用的原因是，不同系统对文件名大小写处理会不同（如 MicroSoft 的 DOS, Windows 系统不区分大小写，但是 Unix / Linux, Mac 系统则默认区分）。

好的命名举例：
`dhcp_user_log.c`

坏的命名举例：
`dhcp_user-log.c`: 不推荐用'-'分隔
`dhcpuserlog.c`: 未分割单词，可读性差

函数命名

函数命名统一使用大驼峰风格。

建议1.3 函数的命名尽量遵循阅读习惯

动作类函数名，一般使用动宾结构。如：

```
AddTableEntry() // OK
DeleteUser()    // OK
GetUserInfo()   // OK
```

判断型函数，可以用形容词，或加 is：

```
DataReady()     // OK
IsRunning()     // OK
JobDone()       // OK
```

数据型函数：

```
TotalCount()    // OK
GetTotalCount() // OK
```

变量命名

变量命名使用小驼峰风格，包括全局变量，局部变量，函数声明或定义中的参数，带括号宏中的参数。

规则1.3 全局变量应增加 'g_' 前缀，函数内静态变量命名不需要加特殊前缀

全局变量是应当尽量少使用的，使用时应特别注意，所以加上前缀用于视觉上的突出，促使开发人员对这些变量的使用更加小心。
全局静态变量命名与全局变量相同，函数内的静态变量命名与普通局部变量相同。

```
int g_activeConnectCount;

void Func()
{
    static int pktCount = 0;
    ...
}
```

建议1.4 在能够表达相关含义的前提下，局部变量尽可能简短

函数局部变量的命名，在能够表达相关含义的前提下，应该简短。

如下：

```
int Func(...)
{
    enum PowerBoardStatus powerBoardStatusOfSlot; // Not good: 局部变量有点长
    powerBoardStatusOfSlot = GetPowerBoardStatus(slot);
    if (powerBoardStatusOfSlot == POWER_OFF) {
        ...
    }
    ...
}
```

更好的写法：

```
int Func(...)
{
    enum PowerBoardStatus status;   // Good: 结合上下文，status 已经能明确表达意思
    status = GetPowerBoardStatus(slot);
    if (status == POWER_OFF) {
        ...
    }
    ...
}
```

类似的， tmp 可以用来称呼任意类型的临时变量。
过短的变量命名应慎用，但有时候，单字符变量也是允许的，如用于循环语句中的计数器变量：

```
int i;
...
for (i = 0; i < COUNTER_RANGE; i++) {
    ...
}
```

或一些简单的数学计算函数中的变量：

```
int Mul(int a, int b)
{
    return a * b;
}
```

类型命名

类型命名采用大驼峰命名风格。
类型包括结构体、联合体、枚举类型名。

例:

```
struct MsgHead {
    enum MsgType type;
    int msgLen;
    char *msgBuf;
};

union Packet {
    struct SendPacket send;
    struct RecvPacket recv;
};

enum BaseColor {
    RED,    // 注意，枚举类型是大驼峰，枚举值应使用宏风格
    GREEN,
    BLUE
};
```

通过 typedef 对结构体、联合体、枚举起别名时，尽量使用匿名类型。
若需要指针自嵌套，可以增加 'tag' 前缀或下划线后缀。

```
typedef struct {    // Good: 无须自嵌套，使用匿名结构体
    int a;
    int b;
} MyType;           // 结构体别名用大驼峰风格
```

```
typedef struct tagNode {    // Good: 使用 tag 前缀。这里也可以使用 'Node_'代替也可以。
    struct tagNode *prev;
    sutrct tagNode *next;
} Node;             // 类型主体用大驼峰风格
```

通过 typedef / #define 对基本类型起别名时，可以遵循C语言本身已有参考实例。如"stdint.h"

```
typedef void *Handle;           // OK: 大驼峰
typedef unsigned int uint32;    // OK: 不是大驼峰，但符合类似 int 风格
```

建议1.5 避免滥用 typedef/#define 对基本类型起别名

除有明确的必要性，否则不要用 typedef/#define 对基本数值类型进行重定义。

必要性指：

- 屏蔽不同平台（CPU/OS）下，C基本数值类型的位宽差异

```
#if __WORDSIZE == 64
typedef unsigned long int   uintptr;
#else
typedef unsigned int        uintptr;
#endif
```

注意：
应由平台、产品或独立模块统一定义一套基本类型别名，以屏蔽不同平台下的类型位宽差异，不要私自定义。
如未定义，可以考虑直接使用C99标准类型(stdint.h)。
当整合其他独立模块代码（如开源代码、第三方代码）时，可增加适配层隔离定义冲突。

- 可预见的未来需要提高精度

```
typedef uint8 DevId;
...
// 若干版本后扩展成 16-bit
typedef uint16 DevId;
```

- 有特殊作用的类型

```
typedef void *Handle;
```

注意：**能用 typedef 的地方，尽量不用 #define 进行别名定义**

除上述理由外，应避免给基本数值类型别名定义。因为类型别名可读性并不好，隐藏了基本数值类型信息，如位宽，是否带符号。
滥用举例：

```
typedef u16 MyCounter;
...
int Foo(...)
{
    MyCounter c;
    ...
    while (c >= 0) {    // Wrong: 这里条件恒成立
        (void)printf("counter = %d\n", c);  // Wrong: 这里应该是 "%u"
        ...
    }
    ...
}
```

对'MyCounter'是否可能小于0，打印时用'%d'还是'%u'都不是很直观，极容易引入上述类似缺陷。

宏、常量、枚举命名

宏、枚举值采用全大写，下划线连接的格式。
全局作用域内的 const 常量全大写，下划线连接；函数局部 const 常量，使用小驼峰命名风格。

函数式宏，如果功能上可以替代函数，也可以与函数的命名方式相同，使用大驼峰命名风格。
这种做法会让宏与函数看起来一样，容易混淆，需要特别注意。

示例：

```
#define PI 3.14
#define MAX(a, b)   (((a) < (b)) ? (b) : (a))
```

```
enum BaseColor {    // 注意，枚举类型名用大驼峰，其下面的取值是全大写，下划线相连
    RED,
    GREEN,
    BLUE
};

const int VERSION = 200;        // 全局常量

int Func(...)
{
    const int bufSize = 100;    // 函数局部常量
    void *p = malloc(bufSize);
    ...
}

#ifdef SOME_DEFINE
void Bar(int);
#define Foo(a) Bar(a)           // 特殊场景，用大驼峰风格命名函数式宏
#else
int Foo(int);
#endif
```

建议1.6 避免函数式宏中的临时变量命名污染外部作用域

首先,**尽量少的使用函数式宏。**

当函数式宏需要定义局部变量时，为了防止跟外部函数中的局部变量有命名冲突。

后置下划线，是一种解决方案。 例：

```
#define SWAP_INT(a, b) do { \
    int tmp_ = a; \
    a = b; \
    b = tmp_; \
} while (0)
```

2 排版格式

尽管有些编程的排版风格因人而已，但是我们强烈建议和要求在同一个项目中使用统一的编码风格，以便所有人都能够轻松的阅读和理解代码，增强代码的可维护性。
对于C和C++编程语言，可以使用 clang-format 进行代码自动化排版，以便在同一个开发团队内部形成统一的编码风格。

行宽

建议2.1 行宽不超过 120 个字符

代码行宽不宜过长，否则不利于阅读。
强烈建议和要求每行字符数不要超过 **120** 个；除非超过 **120** 能显著增加可读性，并且不会隐藏信息。 虽然现代显示器分辨率已经很高，但是行宽过长，反而提高了阅读理解的难度；跟本规范提倡的“清晰”“简洁”原则相背。

理由:

- 间接的引导开发去缩短函数、变量的命名，减少嵌套的层数，提升可读性；
- 保持和绝大多数开源代码中的行宽一致，和主流编码规范（Google、Linux Kernel、微软）保持一致；
- 方便代码按照行进行差异对比。

如下场景不宜换行，可以例外：

- 换行会导致内容截断，无法被方便查找(grep)的字符串，如命令行或 URL 等等。包含这些内容的代码或注释，可以适当例外。
- #include / #error 语句可以超出行宽要求，但是也需要尽量避免。

例：

```
#ifndef XXX_YYY_ZZZ
#error Header aaaa/bbbb/cccc/abc.h must only be included after xxxx/yyyy/zzzz/xyz.h
#endif
```

缩进

规则2.1 使用空格进行缩进，每次缩进4个空格

只允许使用空格(space)进行缩进，每次缩进为 **4** 个空格。不允许使用Tab键进行缩进。
当前几乎所有的集成开发环境（IDE）和代码编辑器都支持配置将Tab键自动扩展为**4**空格输入，请配置你的代码编辑器支持使用空格进行缩进。

大括号

规则2.2 大括号使用 K&R 对齐风格

K&R风格
函数左大括号另起一行放行首，并独占一行；其他左大括号跟随语句放行末。
右大括号独占一行，除非后面跟着同一语句的剩余部分，如 do 语句中的 while，或者 if 语句的 else/else if，或者逗号、分号。

如：

```
struct MyType {      // Good: 跟随语句放行末，前置1空格
    ...
};                   // Good: 右大括号后面紧跟分号

int Foo(int a)
{                    // Good: 函数左大括号独占一行，放行首
    if (...) {
        ...
    } else {         // Good: 右大括号与 else 语句在同一行
        ...
    }                // Good: 右大括号独占一行
}
```

推荐这种风格的理由：

- 代码更紧凑；
- 相比另起一行，放行末使代码阅读节奏感上更连续；
- 符合后来语言的习惯，符合业界主流习惯；
- 现代代码编辑器都具有代码缩进对齐显示的辅助功能，大括号放在行尾并不会对缩进和范围产生理解上的影响。

函数声明和定义

规则2.3 函数声明和定义的返回类型和函数名在同一行；函数参数列表换行时应合理对齐

在声明和定义函数的时候，函数的返回值类型应该和函数名在同一行；如果行宽度允许，函数参数也应该放在一行；否则，函数参数应该换行，并进行合理对齐。
参数列表的左圆括号总是和函数名在同一行，不要单独一行；右圆括号总是跟随最后一个参数。

换行举例：

```
ReturnType FunctionName(ArgType paramName1, ArgType paramName2)   // Good：全在同一行
{
    ...
}

ReturnType VeryVeryVeryLongFunctionName(ArgType paramName1,     // 行宽不满足所有参数，进行换行
                                        ArgType paramName2,     // Good：和上一行参数对齐
                                        ArgType paramName3)
{
    ...
}

ReturnType LongFunctionName(ArgType paramName1, ArgType paramName2, // 行宽限制，进行换行
    ArgType paramName3, ArgType paramName4, ArgType paramName5)     // Good: 换行后 4 空格缩进
{
    ...
}
```

```
ReturnType ReallyReallyReallyReallyLongFunctionName(            // 行宽不满足第1个参数，直接换行
    ArgType paramName1, ArgType paramName2, ArgType paramName3) // Good: 换行后 4 空格缩进
{
    ...
}
```

函数调用

规则2.4 函数调用参数列表应放在一行，超出行宽换行时，保持参数进行合理对齐

函数调用时，函数参数列表放在一行。参数列表如果超过行宽，需要换行并进行合理的参数对齐。
左圆括号总是跟函数名，右圆括号总是跟最后一个参数。

换行举例：

```
ReturnType result = FunctionName(paramName1, paramName2);   // Good：函数参数放在一行

ReturnType result = FunctionName(paramName1,
                                 paramName2,                // Good：保持与上方参数对齐
                                 paramName3);

ReturnType result = FunctionName(paramName1, paramName2,
    paramName3, paramName4, paramName5);                    // Good：参数换行，4 空格缩进

ReturnType result = VeryVeryVeryLongFunctionName(           // 行宽不满足第1个参数，直接换行
    paramName1, paramName2, paramName3);                    // 换行后，4 空格缩进
```

如果函数调用的参数存在内在关联性，按照可理解性优先于格式排版要求，对参数进行合理分组换行。

```
// Good：每行的参数代表一组相关性较强的数据结构，放在一行便于理解
int result = DealWithStructureLikeParams(left.x, left.y,     // 表示一组相关参数
                                         right.x, right.y);  // 表示另外一组相关参数
```

条件语句

规则2.5 条件语句必须要使用大括号

我们要求条件语句都需要使用大括号，即便只有一条语句。
理由：

- 代码逻辑直观，易读；
- 在已有条件语句代码上增加新代码时不容易出错；
- 对于在条件语句中使用函数式宏时，没有大括号保护容易出错（如果宏定义时遗漏了大括号）。

```
if (objectIsNotExist) {          // Good：单行条件语句也加大括号
    return CreateNewObject();
}
```

规则2.6 禁止 if/else/else if 写在同一行

条件语句中，若有多个分支，应该写在不同行。

如下是正确的写法：

```
if (someConditions) {
    ...
} else {    // Good: else 与 if 在不同行
    ...
}
```

下面是不符合规范的案例：

```
if (someConditions) { ... } else { ... } // Bad: else 与 if 在同一行
```

循环

规则2.7 循环语句必须使用大括号

和条件表达式类似，我们要求for/while循环语句必须加上大括号，即便循环体是空的，或循环语句只有一条。

```
for (int i = 0; i < someRange; i++) {   // Good: 使用了大括号
    DoSomething();
}
```

```
while (condition) { }   // Good：循环体是空，使用大括号
```

```
while (condition) {
    continue;           // Good：continue 表示空逻辑，使用大括号
}
```

坏的例子：

```
for (int i = 0; i < someRange; i++)
    DoSomething();      // Bad: 应该加上括号
```

```
while (condition);      // Bad：使用分号容易让人误解是while语句中的一部分
```

switch语句

规则2.8 switch 语句的 case/default 要缩进一层

switch 语句的缩进风格如下：

```
switch (var) {
    case 0:             // Good: 缩进
        DoSomething1(); // Good: 缩进
        break;
    case 1: {           // Good: 带大括号格式
        DoSomething2();
        break;
    }
    default:
        break;
}
```

```
switch (var) {
case 0:                 // Bad: case 未缩进
    DoSomething();
    break;
default:                // Bad: default 未缩进
    break;
}
```

表达式

建议2.2 表达式换行要保持换行的一致性，运算符放行末

较长的表达式，不满足行宽要求的时候，需要在适当的地方换行。一般在较低优先级运算符或连接符后面截断，运算符或连接符放在行末。
运算符、连接符放在行末，表示“未结束，后续还有”。

例：

```
// 假设下面第一行已经不满足行宽要求
if ((currentValue > MIN) &&  // Good：换行后，布尔操作符放在行末
    (currentValue < MAX)) {
    DoSomething();
    ...
}

int result = reallyReallyLongVariableName1 +    // Good: 加号留在行末
             reallyReallyLongVariableName2;
```

表达式换行后，注意保持合理对齐，或者4空格缩进。参考下面例子

```
int sum = longVaribleName1 + longVaribleName2 + longVaribleName3 +
    longVaribleName4 + longVaribleName5 + longVaribleName6;         // OK: 4空格缩进

int sum = longVaribleName1 + longVaribleName2 + longVaribleName3 +
          longVaribleName4 + longVaribleName5 + longVaribleName6;   // OK: 保持对齐
```

变量赋值

规则2.9 多个变量定义和赋值语句不允许写在一行

每行最好只有一个变量初始化的语句，更容易阅读和理解。

```
int maxCount = 10;
bool isCompleted = false;
```

下面是不符合规范的示例：

```
int maxCount = 10; bool isCompleted = false; // Bad：多个初始化放在了同一行
int x, y = 0;  // Bad：多个变量定义需要分行，每行一个

int pointX;
int pointY;
...
pointX = 1; pointY = 2;  // Bad：多个变量赋值语句放同一行
```

例外情况：
对于多个相关性强的变量定义，且无需初始化时，可以定义在一行，减少重复信息，以便代码更加紧凑。

```
int i, j;  // Good：多变量定义，未初始化，可以写在一行
for (i = 0; i < row; i++) {
    for (j = 0; j < col; j++) {
        ...
    }
}
```

初始化

初始化包括结构体、联合体、及数组的初始化

规则2.10 初始化换行时要有缩进，并进行合理对齐

结构体或数组初始化时，如果换行应保持4空格缩进。
从可读性角度出发，选择换行点和对齐位置。

```
// Good: 满足行宽要求时不换行
int arr[4] = { 1, 2, 3, 4 };

// Good: 行宽较长时，换行让可读性更好
const int rank[] = {
    16, 16, 16, 16, 32, 32, 32, 32,
    64, 64, 64, 64, 32, 32, 32, 32
};
```

对于复杂结构数据的初始化，尽量清晰、紧凑。
参考如下格式：

```
int a[][4] = {
    { 1, 2, 3, 4 }, { 2, 2, 3, 4 }, // OK.
    { 3, 2, 3, 4 }, { 4, 2, 3, 4 }
};

int b[][8] = {
    { 1, 2, 3, 4, 5, 6, 7, 8 },     // OK.
    { 2, 2, 3, 4, 5, 6, 7, 8 },
};
```

```
int c[][8] = {
    {
        1, 2, 3, 4, 5, 6, 7, 8      // OK.
    }, {
        2, 2, 3, 4, 5, 6, 7, 8
    }
};
```

规则2.11 结构和联合体在按成员初始化时，每个成员初始化单独一行

C99标准支持结构体和联合体按照成员进行初始化，标准中叫"指定初始化"（designated initializer）。 如果按照这种方式进行初始化，每个成员的初始化单独一行。

```
struct Date {
    int year;
    int month;
    int day;
};

struct Date date = {    // Good：使用指定初始化方式时，每行初始化一个
    .year   = 2000,
    .month  = 1,
    .day    = 1
};
```

指针

建议2.3 指针类型"*"跟随变量名或者类型，不要两边都留有空格或都没有空格

声明或定义指针变量或者返回指针类型函数时，"*" 靠左靠右都可以，但是不要两边都有或者都没有空格。

```
int *p1;    // OK.
int* p2;    // OK.

int*p3;     // Bad: 两边都没空格
int * p4;   // Bad: 两边都有空格
```

例外：
当需要一行同时声明多个变量时，"*" 跟随变量。

```
int *p, *q; // OK.
int* a, b;  // Bad: 很容易将 b 误理解成指针
```

当变量被 const 或 restrict 修饰时，"*" 无法跟随变量，此时也不要跟随类型。

```
char * const VERSION = "V100";
int Foo(const char * restrict p);
```

编译预处理

规则2.12 编译预处理的"#"统一放在行首，嵌套编译预处理语句时，"#"可以进行缩进

编译预处理的"#"统一放在行首；即便编译预处理的代码是嵌入在函数体中的，"#"也应该放在行首。

```
#if defined(__x86_64__) && defined(__GCC_HAVE_SYNC_COMPARE_AND_SWAP_16) // Good：#"放在行首
#define ATOMIC_X86_HAS_CMPXCHG16B 1  // Good：#"放在行首
#else
#define ATOMIC_X86_HAS_CMPXCHG16B 0
#endif

int FunctionName()
{
    if (someThingError) {
        ...
#ifdef HAS_SYSLOG        // Good：即便在函数内部，#"也放在行首
        WriteToSysLog();
#else
        WriteToFileLog();
#endif
    }
}
```

嵌套的预处理语句"#"可以按照缩进要求进行缩进对齐，区分层次。

```
#if defined(__x86_64__) && defined(__GCC_HAVE_SYNC_COMPARE_AND_SWAP_16)
    #define ATOMIC_X86_HAS_CMPXCHG16B 1  // Good：区分层次，便于阅读
#else
    #define ATOMIC_X86_HAS_CMPXCHG16B 0
#endif
```

建议2.4 函数式宏中的'do {'应该放在行尾，'while(0)'和右大括号放在一行

函数式宏将`do {`放在行尾，可以让代码更紧凑。

```
// Good: 'do {' 跟随放在行末
#define FOO(x) do { \
    DoSomething1((x)); \
    DoSomething2((x)); \
} while (0)
```

宏定义中的反斜杠`\`放到行尾，跟主体代码之间留白，上例中留白为一个空格。
也可以将反斜杠对齐。

```
// Good: 反斜杠 '\' 上下对齐
#define FOO(x) do {     \
    DoSomething1((x));  \
    DoSomething2((x));  \
} while (0)
```

空格和空行

规则2.13 水平空格应该突出关键字和重要信息，避免不必要的留白

水平空格应该突出关键字和重要信息，每行代码尾部不要加空格。 总体规则如下：

- if, switch, case, do, while, for等关键字之后加空格；
- 小括号内部的两侧，不要加空格；
- 大括号内部的两侧，要加空格，但内容简单时可以例外 ，如'{}', '{0}', '{1}'等
- 多重括号之间，不要加空格
- 二元操作符（= + - < > * / % | & ^ <= >= == != ）左右两侧加空格
- 一元操作符（& * + - ~ ! ）之后不要加空格；
- 三目运算符（? : ）符号两侧均需要空格
- 前置和后置的自增、自减（++ -- ）和变量之间不加空格
- 结构体成员操作符（. -> ）前后不加空格

常规情况：

```
int i = 0;                  // Good：变量初始化时，= 前后应该有空格，分号前面不要留空格
int buf[BUF_SIZE] = {0};    // Good：数组初始化时，大括号内空格可选
int arr[] = { 10, 20 };     // Good: 正常大括号内部两侧需加空格
```

函数定义和函数调用：

```
int result = Foo(arg1,arg2);
                     ^         // Bad: 逗号后面应该有空格

int result = Foo( arg1, arg2 );
                ^          ^   // Bad: 小括号内部两侧不应该有空格
```

指针和取地址

```
x = *p;     // Good：*操作符和指针p之间不加空格
p = &x;     // Good：&操作符和变量x之间不加空格
x = r.y;    // Good：通过.访问成员变量时不加空格
x = r->y;   // Good：通过->访问成员变量时不加空格
```

操作符：

```
x = 0;    // Good：赋值操作的=前后都要加空格
x = -5;   // Good：负数的符号和数值之前不要加空格
++x;      // Good：前置和后置的++/--和变量之间不要加空格
x--;

if (x && !y)        // Good：布尔操作符前后要加上空格，！操作和变量之间不要空格
v = w * x + y / z;  // Good：二元操作符前后要加空格
v = w * (x + z);    // Good：括号内的表达式前后不需要加空格
```

循环和条件语句：

```
if (condition) {  // Good：if关键字和括号之间加空格，括号内条件语句前后不加空格
    ...
} else {          // Good：else关键字和大括号之间加空格
    ...
}

while (condition) {}   // Good：while关键字和括号之间加空格，括号内条件语句前后不加空格

for (int i = 0; i < someRange; ++i) {  // Good：for关键字和括号之间加空格，分号之后加空格
    ...
}

switch (var) {  // Good: switch 关键字后面有1空格
    case 0:     // Good：case语句条件和冒号之间不加空格
        ...
        break;
```

```
    ...
    default:
        ...
        break;
}
```

注意：当前的集成开发环境（IDE）和代码编辑器都可以设置删除行尾的空格，请正确配置你的编辑器。

建议2.5 空行要尽可能的少，保持代码紧凑

空行要尽可能的少，以便于显示更多的代码，便于代码理解。下面有一些建议遵守的规则：

- 根据上下内容的相关程度，合理安排空行；
- 相邻的两个函数定义之间的间隔使用 **1** 个空行；
- 大括号内的代码块行首之前和行尾之后不要加空行。

```
int Foo()
{
    ...
}


// Bad：两个函数定义间应该使用一个空行而不是两个
int Bar()
{
    ...
}


if (...) {
        // Bad：大括号内的代码块行首不要加入空行
    ...
        // Bad：大括号内的代码块行尾不要加入空行
}

int Foo(...)
{
        // Bad：函数体内行首不要加空行
    ...
}
```

3 注释

一般的，尽量通过清晰的架构逻辑，好的符号命名来提高代码可读性；需要的时候，才辅以注释说明。
注释是为了帮助阅读者快速读懂代码，所以要从读者的角度出发，**按需注释**。

注释内容要简洁、明了、无二义性，信息全面且不冗余。

注释跟代码一样重要。
写注释时要换位思考，用注释去表达此时读者真正需要的信息。在代码的功能、意图层次上进行注释，即注释解释代码难以表达的意图，不要重复代码信息。
修改代码时，也要保证其相关注释的一致性。只改代码，不改注释是一种不文明行为，破坏了代码与注释的一致性，让阅读者迷惑、费解，甚至误解。

注释风格

在 C 代码中，使用 `/*` `*/` 和 `//` 都是可以的。
按注释的目的和位置，注释可分为不同的类型，如文件头注释、函数头注释、代码注释等等；
同一类型的注释应该保持统一的风格。

注意：本文示例代码中，大量使用 '//' 后置注释只是为了更精确的描述问题，并不代表这种注释风格更好。

文件头注释

规则3.1 文件头注释必须包含版权许可、功能说明、作者和创建日期

文件头注释必须按顺序包含上述四项。
如果需要在文件头注释中增加其他内容，可以后面以相同格式补充。
比如：注意事项、修改历史、等等。

版权许可内容及格式必须如下，中文版：
`版权所有（c）华为技术有限公司 2012-2018`
英文版：
`Copyright (c) Huawei Technologies Co., Ltd. 2012-2018. All rights reserved.`

其中，2012-2018 根据实际需要可以修改， 2012 是文件首次创建年份，而 2018 是最后文件修改年份。
对文件有重大修改时，必须更新后面年份。

文件头注释必须从文件顶头开始。项目内保持统一格式。
具体格式由项目或更大范围统一制定，格式可参考：

```
/*
 * Copyright (c) Huawei Technologies Co., Ltd. 2012-2018. All rights reserved.
 * Description: 文件功能描述
 * Author: 王二 w00123456
 * Create: 2012-12-22
 */
```

增加包含了 'Note' 和 'History 的文件头格式如下：

```
/*
 * Copyright (c) Huawei Technologies Co., Ltd. 2012-2018. All rights reserved.
 * Description: 文件功能描述
 * Author: 王二 w00123456
 * Create: 2012-12-22
 * Notes: 特别需要注意的信息
 * History: 2018-01-01 张三 z00123457 某特性扩展
 *          2019-01-01 李四 l00123458 某重大修改
 */
```

编写文件头注释应注意：

- 必须包含必选项
- 禁止空有格式，无内容。

  如上述例子，如果选项 'Notes' 后面无内容，则应整行删除。
- 若内容过长，超出行宽要求，换行时应注释对齐。

  对齐可参考上述例子 'History'

函数头注释

规则3.2 函数命名无法表达的信息，必须加函数头注释辅助说明; 禁止空有格式的函数头

函数头注释统一放在函数声明或定义上方。
选择使用如下风格之一：
使用'//'写函数头

```
// 单行函数头
int Func1(void);

// 多行函数头
// 第二行
int Func2(void);
```

使用'/*' '*/' 写函数头

```
/* 单行函数头 */
int Func1(void);

/*
 * 单行或多行函数头
 * 第二行
 */
int Func2(void);
```

函数尽量通过函数名自注释，**按需**写函数头注释。
不要写无用、信息冗余的函数头；不要写空有格式的函数头。

函数头注释内容**可选**，但不限于：功能说明、返回值，性能约束、用法、内存约定、算法实现、可重入的要求等等。
模块对外头文件中的函数接口声明，其函数头注释，应当将重要、有用的信息表达清楚。

例：

```
/*
 * 返回实际写入的字节数，-1表示写入失败
 * 注意，内存 buf 由调用者负责释放
 */
int WriteString(char *buf, int len);
```

坏的例子：

```
/*
 * 函数名：WriteString
 * 功能：写入字符串
 * 参数：
 * 返回值：
 */
int WriteString(char *buf, int len);
```

上面例子中的问题：

- 参数、返回值，空有格式没内容
- 函数名信息冗余
- 关键的 buf 由谁释放没有说清楚

代码注释

规则3.3 代码注释放于对应代码的上方或右边

规则3.4 注释符与注释内容间要有1空格；右置注释与前面代码至少1空格

代码上方的注释，应该保持对应代码一样的缩进。
选择并统一使用如下风格之一：
使用'//'

```
// 这是单行注释
DoSomething();

// 这是多行注释
// 第二行
DoSomething();
```

使用'/*' '*/'

```
/* 这是单行注释 */
DoSomething();

/*
 * 这是的单/多行注释
 * 第二行
 */
DoSomething();
```

代码右边的注释，与代码之间，至少留1空格，建议不超过4空格。
通常使用扩展后的 TAB 键即可实现 1-4 空格的缩进。

选择并统一使用如下风格之一：

```
int foo = 100;  // 放右边的注释
int bar = 200;  /* 放右边的注释 */
```

右置格式在适当的时候，上下对齐会更美观。
对齐后的注释，离左边代码最近的那一行，保证1-4空格的间隔。
例：

```
#define A_CONST 100         /* 相关的同类注释，可以考虑上下对齐 */
#define ANOTHER_CONST 200   /* 上下对齐时，与左侧代码保持间隔 */
```

当右置的注释超过行宽时，请考虑将注释置于代码上方。

规则3.5 不用的代码段直接删除，不要注释掉

被注释掉的代码，无法被正常维护；当企图恢复使用这段代码时，极有可能引入易被忽略的缺陷。
正确的做法是，不需要的代码直接删除掉。若再需要时，考虑移植或重写这段代码。

这里说的注释掉代码，包括用 /* */ 和 //，还包括 #if 0， #ifdef NEVER_DEFINED 等等。

建议3.1 正式交付给客户的代码不能包含 TODO/TBD/FIXME 注释

TODO/TBD 注释一般用来描述已知待改进、待补充的修改点
FIXME 注释一般用来描述已知缺陷
它们都应该有统一风格，方便文本搜索统一处理。如：

```
// TODO(<author-name>): 补充XX处理
// FIXME: XX缺陷
```

在版本开发阶段，可以使用此类注释用于突出标注；交付前应该全部处理并删除掉。

建议3.2 case语句块结束时如果不加break，需要有注释说明(fall-through)

有时候需要对多个case标签做相同的事情，case语句在结束不加break，直接执行下一个case标签中的语句，这在C语法中称之为"fall-through"。
这种情况下，需要在"fall-through"的地方加上注释，清晰明确的表达出这样做的意图；或者至少显式指明是 "fall-through"。

例，显式指明 fall-through：

```
switch (var) {
    case 0:
        DoSomething();
        /* fall-through */
    case 1:
        DoSomeOtherThing();
        ...
        break;
    default:
        DoNothing();
        break;
}
```

如果 case 语句是空语句，则可以不用加注释特别说明:

```
switch (var) {
    case 0:
    case 1:
        DoSomething();
        break;
    default:
        DoNothing();
        break;
}
```

注释语言

建议3.3 使用流利的中文或英文进行注释

使用团队内最擅长、沟通效率最高的语言写注释；仅限中文或英文。
注释语言由开发团队统一决定。

注释要求无错别字(拼写错误)、语句通顺（语法正确）。

4 头文件

对于C语言来说，头文件的设计体现了大部分的系统设计。
正确使用头文件可使代码在可读性、文件大小和编译构建性能上大为改观。

本章从编程规范的角度总结了一些方法，可用于帮助合理规划头文件。

头文件职责

头文件是模块或文件的对外接口。
头文件中适合放置接口的声明，不适合放置实现（内联函数除外）。
头文件应当职责单一。头文件过于复杂，依赖过于复杂还是导致编译时间过长的主要原因。

规则4.1 每一个.c文件都应该有相应的.h文件，用于声明需要对外公开的接口

通常情况下，每个.c文件都有一个相应的.h，用于放置对外提供的函数声明、宏定义、类型定义等。
如果一个.c文件不需要对外公布任何接口，则其就不应当存在。

例外：程序的入口（如main函数所在的文件），单元测试代码，动态库代码。

示例:
foo.h 内容

```
#ifndef FOO_H
#define FOO_H

int Foo();  // Good：头文件中声明对外接口

#endif
```

foo.c 内容

```
static void Bar();  // Good: 对内函数的声明放在.c文件的头部，并声明为static限制其作用域

void Foo()
{
    Bar();
}

void Bar()
{
    // Do something;
}
```

内部使用的函数声明，宏、枚举、结构体等定义不应放在头文件中。

有些产品中，习惯一个.c文件对应两个.h文件，一个用于存放对外公开的接口，一个用于存放内部需要用到的定义、声明等，以控制.c文件的代码行数。
不提倡这种风格，产生这种风格的根源在于.c过大，应当首先考虑拆分.c文件。
另外，一旦把私有定义、声明放到独立的头文件中，就无法从技术上避免别人包含。

本规则反过来并不一定成立。比如：
有些特别简单的头文件，如命令 ID 定义头文件，不需要有对应的.c存在。
同一套接口协议下，有多个实例，由于接口相同且稳定，所以允许出现一个.h对应多个.c文件。

建议4.1 一个模块包含的多个.c文件，建议放在同一个目录下，为方便外部使用者，建议一个模块提供一个.h，声明模块整体对外提供的接口

需要注意的是，这个.h并不是简单的包含所有内部的.h，它是为了模块使用者的方便，对外整体提供的模块接口。

优点：即使以后模块的内部实现改变了，比如把一个源文件拆成两个源文件，使用者也不必关心，甚至如果对外功能不变，连重新编译都不需要。

比如 GTest，作为一个整体对外提供 C++ 单元测试框架，其1.5版本的 gtest 工程下有6个源文件和12个头文件。但是它对外只提供一个 gtest.h，只要包含 gtest.h 即可使用 GTest 提供的所有对外提供的功能，使用者不必关心 GTest 内部各个文件的关系。

对于有些模块，其内部功能相对松散，可能并不一定需要提供这个.h，而是直接提供各个子模块的头文件。

建议4.2 头文件的扩展名只使用.h，不使用非习惯用法的扩展名，如.inc

有些产品中使用了 .inc 作为头文件扩展名，这不符合C语言的习惯用法。 在使用 .inc 作为头文件扩展名的产品，习惯上用于标识此头文件为私有头文件。 但是从产品的实际代码来看，这一条并没有被遵守，一个 .inc 文件被多个 .c 包含。 本规范不提倡将私有定义单独放在头文件中，具体见规则4.1。

除此之外，使用.inc还导致 Source Insight、Visual Stduio 等 IDE 工具无法识别其为头文件，导致很多功能不可用，如“跳转到变量定义处”。 虽然可以通过配置，强迫 IDE 识别 .inc 为头文件，但是有些软件无法配置，如 Visual Assist 只能识别 .h 而无法通过配置识别 .inc。

头文件依赖

头文件包含是一种依赖关系，头文件应向稳定的方向包含。
一般来说，应当让不稳定的模块依赖稳定的模块，从而当不稳定的模块发生变化时，不会影响（编译）稳定的模块。

依赖的方向应该是：产品依赖于平台，平台依赖于标准库。

除了不稳定的模块依赖于稳定的模块外，更好的方式是每个模块都依赖于接口，这样任何一个模块的内部实现更改都不需要重新编译另外一个模块。
在这里，假设接口本身是最稳定的。

规则4.2 禁止头文件循环依赖

头文件循环依赖，指 a.h 包含 b.h，b.h 包含 c.h，c.h 包含 a.h， 导致任何一个头文件修改，都导致所有包含了a.h/b.h/c.h的代码全部重新编译一遍。
而如果是单向依赖，如a.h包含b.h，b.h包含c.h，而c.h不包含任何头文件，则修改a.h不会导致包含了b.h/c.h的源代码重新编译。

头文件循环依赖直接体现了架构设计上的不合理，可通过优化架构去避免。

规则4.3 禁止包含用不到的头文件

包含不需要头文件，则引入了不必要的依赖，增加了模块或单元之间的耦合度，增加了代码复杂性，可维护性差。

很多系统中头文件包含关系复杂。开发人员为了省事起见，直接包含一切想到的头文件，甚至发布了一个 god.h，其中包含了所有头文件，然后发布给各个项目组使用。
这种只图一时省事的做法，不仅导致整个系统的编译时间恶化，而且代码的维护成本非常高。

规则4.4 头文件应当自包含

简单的说，自包含就是任意一个头文件均可独立编译。如果一个文件包含某个头文件，还要包含另外一个头文件才能工作的话，给这个头文件的用户增添不必要的负担。

比如，如果a.h不是自包含的，需要包含b.h才能编译，会带来的危害： 每个使用a.h头文件的.c文件，为了让引入的a.h的内容编译通过，都要包含额外的头文件b.h。 额外的头文件b.h必须在a.h之前进行包含，这在包含顺序上产生了依赖。

注意: 该规则需要与规则4.3 禁止包含用不到的头文件规则一起使用，不能为了让a.h自包含，而在a.h中包含不必要的头文件。
a.h要刚刚可以自包含，不能在a.h中多包含任何满足自包含之外的其他头文件。

规则4.5 头文件必须编写#define保护，防止重复包含

为防止头文件被多重包含，所有头文件都应当使用 #define 作为包含保护；不要使用 #pragma once

定义包含保护符时，应该遵守如下规则：

- 保护符使用唯一名称；通常为项目源代码树顶层以下的文件路径
- 不要在受保护部分的前后放置代码或者注释，文件头注释除外。

假定 VOS 工程的 timer 模块的 timer.h，其目录为 `vos/include/timer.h`。其保护符若使用 'TIME_H' 很容易不唯一，所以使用项目源代码树的全路径，如：

```
#ifndef VOS_INCLUDE_TIMER_H
#define VOS_INCLUDE_TIMER_H

...

#endif
```

规则4.6 禁止通过 extern 声明的方式引用外部函数接口、变量

只能通过包含头文件的方式使用其他模块或文件提供的接口。
通过 extern 声明的方式使用外部函数接口、变量，容易在外部接口改变时可能导致声明和定义不一致。
同时这种隐式依赖，容易导致架构腐化。

不符合规范的案例：
a.c 内容

```
extern int Foo();   // Bad: 通过 extern 的方式引用外部函数

void Bar()
{
    int i = Foo();  // 这里使用了外部接口 Foo
    ...
}
```

应该改为：
a.c 内容

```
#include "b.h"      // Good: 通过包含头文件的方式使用其他.c提供的接口

void Bar()
{
    int i = Foo();
    ...
}
```

b.h 内容

```
int Foo();
```

b.c内容

```
int Foo()
{
    // Do something
}
```

例外，有些场景需要引用其内部函数，但并不想侵入代码时，可以 extern 声明方式引用。
如：
针对某一内部函数进行单元测试时，可以通过 extern 声明来引用被测函数；
当需要对某一函数进行打桩、打补丁处理时，允许 extern 声明该函数。

规则4.7 禁止在 extern "C" 中包含头文件

在 extern "C" 中包含头文件，有可能会导致 extern "C" 嵌套，部分编译器对 extern "C" 嵌套层次有限制，嵌套层次太多会编译错误。

extern "C" 通常出现在 C，C++ 混合编程的情况下，在 extern "C" 中包含头文件，可能会导致被包含头文件的原有意图遭到破坏，比如链接规范被不正确地更改。

示例，存在a.h和b.h两个头文件：
a.h 内容

```
...
#ifdef __cplusplus
void Foo(int);
#define A(value) Foo(value)
#else
void A(int)
#endif
```

b.h 内容

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

#include "a.h"
void B();

#ifdef __cplusplus
}
#endif
```

使用C++预处理器展开b.h，将会得到

```
extern "C" {
    void Foo(int);
    void B();
}
```

按照 a.h 作者的本意，函数 Foo 是一个 C++ 自由函数，其链接规范为 "C++"。 但在 b.h 中，由于 `#include "a.h"` 被放到了 `extern "C"` 的内部，函数 Foo 的链接规范被不正确地更改了。

例外： 如果在 C++ 编译环境中，想引用纯C的头文件，这些C头文件并没有 `extern "C"` 修饰。非侵入式的做法是，在 `extern "C"` 中去包含C头文件。

建议4.3 头文件包含顺序：首先是.c相应的.h文件，其它头文件按照稳定度排序

使用标准的头文件包含顺序可增强可读性, 避免隐藏依赖，建议的头文件包含顺序：.c文件对应的.h, C标准库, 系统库的.h, 其他库的.h, 本项目内其他的.h。

举例，foo.c中包含头文件的次序如下:

```
#include "foo/foo.h"

#include <stdlib.h>
#include <string.h>

#include <linux/list.h>
#include <linux/time.h>

#include "platform/base.h"
#include "platform/struct.h"

#include "project/public/log.h"
```

将foo.h放在最前面可以保证当foo.h遗漏某些必要的库，或者有错误时，foo.c 的构建会立刻中止，减少编译时间。

5 函数

函数的作用：避免重复代码、增加可重用性；分层，降低复杂度、隐藏实现细节，使程序更加模块化，从而更有利于程序的阅读，维护。

函数应该简洁、短小。
一个函数只完成一件事情。

函数设计

函数设计的精髓：编写整洁函数，同时把代码有效组织起来。代码简单直接、不隐藏设计者的意图、用干净利落的抽象和直截了当的控制语句将函数有机组织起来。

规则5.1 避免函数过长，函数不超过50行（非空非注释）

函数应该可以一屏显示完 (50行以内)，只做一件事情，而且把它做好。

过长的函数往往意味着函数功能不单一，过于复杂，或过分呈现细节，未进行进一步抽象。

例外：某些实现算法的函数，由于算法的聚合性与功能的全面性，可能会超过50行。

即使一个长函数现在工作的非常好, 一旦有人对其修改, 有可能出现新的问题, 甚至导致难以发现的bug。
建议将其拆分为更加简短并易于管理的若干函数，以便于他人阅读和修改代码。

规则5.2 避免函数的代码块嵌套过深，不要超过4层

函数的代码块嵌套深度指的是函数中的代码控制块（例如：if、for、while、switch等）之间互相包含的深度。
每级嵌套都会增加阅读代码时的脑力消耗，因为需要在脑子里维护一个“栈”（比如，进入条件语句、进入循环等等）。
应该做进一步的功能分解，从而避免使代码的阅读者一次记住太多的上下文。

使用`卫语句`可以有效的减少 if 相关的嵌套层次。例：
原代码：

```
int Foo(...)
{
    if (received) {
        type = GetMsgType(msg);
        if (type != UNKNOWN) {
            return DealMsg(...);
        }
    }
    return -1;
}
```

使用`卫语句`重构：

```
int Foo(...)
{
    if (!received) {    // Good: 使用'卫语句'
        return -1;
    }

    type = GetMsgType(msg);
    if (type == UNKNOWN) {
        return -1;
    }

    return DealMsg(..);
}
```

建议5.1 对函数的错误返回码要全面处理

一个函数（标准库中的函数/第三方库函数/用户定义的函数）能够提供一些指示错误发生的方法。 这可以通过使用错误标记、特殊的返回数据或者其他手段，不管什么时候函数提供了这样的机制，调用程序应该在函数返回时立刻检查错误指示。

示例：

```
char fileHead[128];
ReadFileHead(fileName, fileHead, sizeof(fileHead)); // Bad: 未检查返回值

DealWithFileHead(fileHead, sizeof(fileHead));       // fileHead 可能无效
```

正确写法：

```
char fileHead[128];
ret = ReadFileHead(fileName, fileHead, sizeof(fileHead));
if (ret != OK) {                // Good: 确保 fileHead 被有效写入
    return ERROR;
}

DealWithFileHead(fileHead, sizeof(fileHead));       // 处理文件头
```

注意，当函数返回值被大量的显式(void)忽略掉时，应当考虑函数返回值的设计是否合理。
如果所有调用者都不关注函数返回值时，请将函数设计成`void`型。

建议5.2 设计函数时，优先使用返回值而不是输出参数

使用返回值而不是输出参数，可以提高可读性，并且通常提供相同或更好的性能。

函数名为 GetXxx、FindXxx 或直接名词作函数名的函数，直接返回对应对象，可读性更好。

建议5.3 定义函数时，参数顺序为：输入参数在前，输出参数在后

首先将参数分组，然后按照输入，输出的顺序排列。在参数组内，按照能够帮助程序员输入正确值的原则来将参数排序。
比如，如果一个函数带有2个参数，“left”和“right”，将“left”置于“right”之前，则它们的放置顺序符合其参数名。当设计一系列具有相同参数的函数时，在各函数内使用一致的顺序。

在加入新参数时不要因为它们是新参数就置于参数列表最后，而是仍然要按照上述的顺序，即将新的输入参数也置于输出参数之前。

符合既成习惯的，应该遵循已有实例参考，安排参数顺序。
如：
格式化函数，应这样 `XxxPrintf(buf, bufLen, "<format string>", param1, param2, ...)`;
拷贝函数，应这样 `XxxCpy(dst, src)`

函数参数

建议5.4 使用强类型参数，避免使用void*

尽管不同的语言对待强类型和弱类型有自己的观点，但是一般认为c/c++是强类型语言，既然我们使用的语言是强类型的，就应该保持这样的风格。
好处是尽量让编译器在编译阶段就检查出类型不匹配的问题。

使用强类型便于编译器帮我们发现错误，如下代码中注意函数 `FooListAddNode` 的使用：

```
struct FooNode {
    struct List link;
    int foo;
};

struct BarNode {
    struct List link;
    int bar;
}

void FooListAddNode(void *node) // Bad: 这里用 void * 类型传递参数
{
    FooNode *foo = (FooNode *)node;
    ListAppend(&g_fooList, &foo->link);
}

void MakeTheList(...)
{
    FooNode *foo;
    BarNode *bar;
    ...

    FooListAddNode(bar);        // Wrong: 这里本意是想传递参数 foo，但错传了 bar，却没有报错
}
```

上述问题有可能很隐晦，不易轻易暴露，从而破坏性更大。
如果明确 `FooListAddNode` 的参数类型，而不是 `void *`，则在编译阶段就能发现上述问题。

```
void FooListAddNode(FooNode *foo)
{
    ListAppend(&g_fooList, &foo->link);
}
```

例外：某些通用泛型接口，需要传入不同类型指针的，可以用 `void *` 入参。

建议5.5 模块内部函数参数的合法性检查，由调用者负责

对于模块外部传入的参数，必须进行合法性检查，保护程序免遭非法输入数据的破坏。
模块内部函数调用，缺省由调用者负责保证参数的合法性，如果都由被调用者来检查参数合法性，可能会出现同一个参数，被检查多次，产生冗余代码，很不简洁。

由调用者保证入参的合法性，这种契约式编程能让代码逻辑更简洁，可读性更好。
示例：

```
int SomeProc(...)
{
    int data;

    bool dataOK = GetData(&data);       // 获取数据
    if (!dataOK) {                      // 检查上一步结果，其实也就保证了数据合法
        return -1;
    }

    DealWithData(data);                 // 调用数据处理函数
    ...
}

void DealWithData(int data)
{
    if (data < MIN || data > MAX) {     // Bad: 调用者已经保证了数据合法性
        return;
    }

    ...
}
```

建议5.6 函数的参数不应该当作工作变量

在函数体中使用参数作为工作变量，对其进行修改，可能会给函数体内后续代码的理解和维护带来困难和潜在的风险。
需要时，可以使用局部变量来代替它。

如下代码不合理地使用参数作为工作变量：

```
int Foo(int input)
{
    input += Add(input);        // Bad：使用参数作为工作变量，直接修改
    ...
    if (input > threshold) {    // Wrong: 本意是比较参数输入值，但实际其值已经被改变
        DoExtraOperation();
    }
    ...
}
```

正确写法：

```
int Foo(int input)
{
    int workVar = input;        // Good：使用局部变量
    workVar += Add(workVar);
    ...
    if (input > threshold) {    // 参数原始值没变，和期望一致
        DoExtraOperation();
    }
    ...
}
```

也有例外。当入参含义所指数据确实随着函数逻辑执行而改变时，则可以对入参进行改动。
前提是函数足够简短、清晰。例：

```
void Process(int connectCount)
{
    if (...) {
        NewConnect(...);    // 创建新的 connect
        connectCount++;     // OK: 实际 connect 数量已经增加
    }
    ...
    Tell(connectCount);     // OK: 和期望一样
}
```

建议5.7 函数的指针参数如果不是用于修改所指向的对象就应该声明为指向const的指针

const 指针参数，将限制函数通过该指针修改所指向对象，使代码更牢固、安全。

示例：C99标准 7.21.4.4 中strncmp 的例子，不变参数声明为const。

```
int strncmp(const char *s1, const char *s2, size_t n); // Good：不变参数声明为const
```

建议5.8 函数的参数个数不超过5个

函数的参数过多，会使得该函数易于受外部（其他部分的代码）变化的影响，从而影响维护工作。函数的参数过多同时也会增大测试的工作量。

函数的参数个数不要超过5个，如果超过可以考虑:

- 看能否拆分函数
- 看能否将相关参数合在一起，定义结构体

建议5.9 除格式化函数外，不要使用可变长参函数

可变长参函数的处理过程比较复杂容易引入错误，比如参数个数、类型不匹配问题；而且性能也比较低。
使用过多的可变长参函数将导致函数的维护难度大大增加。

内联函数

内联函数是**C99**引入的一种函数优化手段。函数内联能消除函数调用的开销；并得益于内联实现跟调用点代码的合并，编译器有更大的视角，从而完成更多的代码优化。内联函数跟函数式宏比较类似，两者的分析详见建议6.1。

建议5.10 内联函数不超过10行（非空非注释）

将函数定义成内联一般希望提升性能，但是实际并不一定能提升性能。 如果函数体短小，则函数内联可以有效的缩减目标代码的大小，并提升函数执行效率。
反之，函数体比较大，内联展开会导致目标代码的膨胀，特别是当调用点很多时，膨胀得更厉害，反而会降低执行效率。
内联函数规模建议控制在 **10** 行以内。

不要为了提高性能而滥用内联函数。不要过早优化。一般情况，当有实际测试数据证明内联性能更高时，再将函数定义为内联。 对于类似 setter/getter 短小而且调用频繁的函数，可以定义为内联。

规则5.3 被多个源文件调用的内联函数要放在头文件中定义

内联函数是在编译时内联展开，因此要求内联函数定义必须在调用此函数的每个源文件内可见。
如下所示代码，inline.h 只有`SomeInlineFunc`函数的声明而没有定义。other.c包含inline.h，调用`SomeInlineFunc`时无法内联。

inline.h

```
inline int SomeInlineFunc();
```

inline.c

```
inline int SomeInlineFunc()
{
    // 实现代码
}
```

other.c

```
#include "inline.h"
int OtherFunc()
{
    int ret = SomeInlineFunc();
}
```

由于这个限制，多个源文件如果要调用同一个内联函数，需要将内联函数的定义放在头文件中。
gnu89 在内联函数实现上跟**C99**标准有差异，兼容做法是将函数声明为 **static inline**。

6 宏

函数式宏(function-like macro)

函数式宏是指形如函数的宏(示例代码如下所示)，其包含若干条语句来实现某一特定功能。

```
#define ASSERT(x) do { \
    if (!(x)) { \
        printk(KERN_EMERG "assertion failed %s: %d: %s\n", \
               __FILE__, __LINE__, #x); \
        BUG(); \
    } \
} while (0)
```

建议6.1 尽可能使用函数代替函数式宏

定义函数式宏前，应考虑能否用函数替代。对于可替代场景，建议用函数替代宏。
函数式宏的缺点如下：

- 函数式宏缺乏类型检查，不如函数调用检查严格。示例代码见下。
- 宏展开时宏参数不求值，可能会产生非预期结果，详见规则6.1和规则6.3。
- 宏没有独立的作用域，跟控制流语句配合时，可能会产生如规则6.2描述的非预期结果。
- 宏的技巧性太强（参见下面的规则），例如`#`的用法和无处不在的括号，影响可读性。
- 在特定场景下必须用特定编译器对宏的扩展，如 `gcc` 的 `statement expression`，可移植性也不好。
- 宏在预编译阶段展开后，在其后编译、链接和调试时都不可见；而且包含多行的宏会展开为一行。函数式宏难以调试、难以打断点，不利于定位问题。
- 对于包含大量语句的宏，在每个调用点都要展开。如果调用点很多，会造成代码空间的膨胀。

函数式宏缺乏类型检查的示例代码：

```
#define MAX(a, b)   (((a) < (b)) ? (b) : (a))

int Max(int a, int b)
{
    return (a < b) ? b : a;
}

int TestMacro()
{
```

```
    unsigned int a = 1;
    int b = -1;

    (void)printf("MACRO: max of a(%u) and b(%d) is %d\n", a, b, MAX(a, b));
    (void)printf("FUNC : max of a(%u) and b(%d) is %d\n", a, b, Max(a, b));
    return 0;
}
```

由于宏缺乏类型检查，`MAX`中的`a`和`b`的比较提升为无符号数的比较，结果是**a < b**。输出结果是：

```
MACRO: max of a(1) and b(-1) is -1
FUNC : max of a(1) and b(-1) is 1
```

函数没有宏的上述缺点。但是，函数相比宏，最大的劣势是执行效率不高（增加函数调用的开销和编译器优化的难度）。
为此，C99标准引入了内联函数（gcc在标准之前就引入了内联函数）。

内联函数跟宏类似，也是在调用点展开。不同之处在于内联函数是在编译时展开。
内联函数兼具函数和宏的优点：

- 内联函数/函数执行严格的类型检查
- 内联函数/函数的入参求值只会进行一次
- 内联函数就地展开，没有函数调用的开销
- 内联函数比函数优化得更好

对于性能敏感的代码，可以考虑用内联函数代替函数式宏。
函数和内联函数不能完全替代函数式宏，函数式宏在某些场景更适合。
比如，在日志记录场景下，使用带可变参和默认参数的函数式宏更方便：

```
int ErrLog(const char *file, unsigned long line, const char *fmt, ...);
#define ERR_LOG(fmt, ...) ErrLog(__FILE__, __LINE__, fmt, ##__VA_ARGS__)
```

规则6.1 定义宏时，宏参数要使用完备的括号

宏参数在宏展开时只是文本替换，在编译时再求值。文本替换后，宏包含的语句跟调用点代码合并。
合并后的表达式因为运算符的优先级和结合律，可能会导致计算结果跟期望的不同，尤其是当宏参数在一个表达式中时。

如下所示，是一种错误的写法:

```
#define SUM(a, b) a + b         // Bad.
```

下面这样调用宏，执行结果跟预期不符：
`100 / SUM(2, 8)`将扩展成`(100 / 2) + 8`，预期结果则是`100 / (2 + 8)`。
这个问题可以通过将整个表示式加上括号来解决，如下所示：

```
#define SUM(a, b) (a + b)       // Bad.
```

但是这种改法在下面这种场景又有问题：
`SUM(1 << 2, 8)`扩展成`1 << (2 + 8)`（因为`<<`优先级低于`+`），跟预期结果`(1 << 2) + 8`不符。

这个问题可以通过将每个宏参数都加上括号括号解决，如下所示：

```
#define SUM(a, b) (a) + (b)     // Bad.
```

再看看第三种问题场景：`SUM(2, 8) * 10`。扩展后的结果为`(2) + ((8) * 10)`，跟预期结果`(2 + 8) * 10`不符。

综上所述，正确的写法如下：

```
#define SUM(a, b) ((a) + (b))   // Good.
```

但是要避免滥用括号。如下所示，单独的数字加括号毫无意义。

```
#define SOME_CONST  (100)       // Bad: 不应使用括号
```

另外，'#', '##' 作用于宏参数的用法，对应宏参数无需使用括号。

规则6.2 包含多条语句的函数式宏的实现语句必须放在 do-while(0) 中

包含多条语句的函数式宏必须放在 do-while(0) 中；
但用于封装指定初始化（designated initializer）的宏不需要用 do-while(0) 包裹起来。

宏本身没有代码块的概念。当宏在调用点展开后，宏内定义的表达式和变量融合到调用代码中，可能会出现变量名冲突和宏内语句被分割等问题。 通过 do-while(0) 显式为宏加上边界，让宏有独立的作用域，并且跟分号能更好的结合而形成单条语句，从而规避此类问题。

如下所示的宏是错误的用法（为了说明问题，下面示例代码稍不符规范）：

```
// Not Good.
#define FOO(x) \
    (void)printf("arg is %d\n", (x)); \
    DoSomething((x));
```

当像下面示例代码这样调用宏，for循环只执行了宏的第一条语句，宏的后一条语句只在循环结束后执行一次。

```
for (i = 1; i < 10; i++)
    FOO(i);
```

用大括号将`FOO`定义的语句括起来可以解决上面的问题：

```
#define FOO(x) { \
    (void)printf("arg is %d\n", (x)); \
    DoSomething((x)); \
}
```

由于大括号跟分号没有关联。大括号后紧跟的分号，是另外一个语句。
如下示例代码，会出现'悬挂else' 编译报错：

```
if (condition)
    FOO(10);
else
    FOO(20);
```

正确的写法是用 do-while(0) 把执行体括起来，如下所示：

```
// Good.
#define FOO(x) do { \
    (void)printf("arg is %d\n", (x)); \
    DoSomething((x)); \
} while (0)
```

规则6.3 不允许把带副作用的表达式作为参数传递给函数式宏

由于宏只是文本替换，对于内部多次使用同一个宏参数的函数式宏，将带副作用的表达式作为宏参数传入会导致非预期的结果。
如下所示，宏 `SQUARE` 本身没有问题，但是使用时将带副作用的 `a++` 传入导致 `a` 的值在 `SQUARE` 执行后跟预期不符：

```
#define SQUARE(a) ((a) * (a))

int a = 5;
int b;
b = SQUARE(a++);    // Bad: 实际 a 自增加了 2 次
```

`SQUARE(a++)` 展开后为 `((a++) * (a++))` ，变量 `a` 自增了两次，其值为 `7` ，而不是预期的 `6` 。

正确的写法如下所示：

```
b = SQUARE(a);
a++; // 结果：a = 6，只自增了一次。
```

此外，将函数调用作为参数传入，宏展开后，函数会重复调用两次。如果函数执行结果相同，则浪费了一次调用；如果函数两次调用结果不一样，执行结果可能不符合预期。

建议6.2 函数式宏定义中慎用 return、goto、continue、break 等改变程序流程的语句

宏中使用 return、goto、continue、break 等改变流程的语句，虽然能简化代码，但同时也隐藏了真实流程，不易于理解，容易导致资源泄漏等问题。

首先，宏封装 return 容易导致过度封装和使用。
如下代码，`status` 的判断是主干流程的一部分，用宏封装起来后，变得不直观了，阅读时习惯性把 `RETURN_IF` 宏忽略掉了，从而导致对主干流程的理解有偏差。

```
#define LOG_AND_RETURN_IF_FAIL(ret, fmt, ...) do { \
    if ((ret) != OK) {                              \
        (void)ErrLog(fmt, ##__VA_ARGS__);           \
        return (ret);                               \
    }                                               \
} while (0)

#define RETURN_IF(cond, ret) do {                   \
    if (cond) {                                     \
        return (ret);                               \
    }                                               \
} while (0)

ret = InitModuleA(a, b, &status);
LOG_AND_RETURN_IF_FAIL(ret, "Init module A failed!"); // OK.

RETURN_IF(status != READY, ERR_NOT_READY);  // Bad: 重要逻辑不明显

ret = InitModuleB(c);
LOG_AND_RETURN_IF_FAIL(ret, "Init module B failed!"); // OK.
```

而且，`do-while(0)`中不能包含`break`和`continue`语句，否则`break`和`continue`跟`while(0)`结合，结果是退出`while(0)`循环，而不是退出外层循环。

其次，宏封装 return 也容易引发内存泄漏。再看一个例子：

```
#define CHECK_PTR(ptr, ret) do { \
    if ((ptr) == NULL) { \
        return (ret); \
    } \
} while (0)

...

mem1 = MemAlloc(...);
CHECK_PTR(mem1, ERR_CODE_XXX);

mem2 = MemAlloc(...);
CHECK_PTR(mem2, ERR_CODE_XXX);  // Wrong: 内存泄露
```

如果`mem2`申请内存失败了，`CHECK_PTR`会直接返回，而没有释放`mem1`。
除此之外，`CHECK_PTR`宏命名也不好，宏名只反映了检查动作，没有指明结果。只有看了宏实现才知道指针为空时返回失败。

包含 return、goto、continue、break 等改变流程语句的宏命名，务必要体现对应关键字。

综上所述，不推荐宏定义中封装 return、goto、continue、break 等改变程序流程的语句。

建议6.3 函数式宏不超过10行(非空非注释)

函数式宏本身的一大问题是比函数更难以调试和定位，特别是宏过长，调试和定位的难度更大。
而且宏扩展会导致目标代码的膨胀。建议函数式宏不要超过10行。

7 变量

在C语言编码中，除了函数，最重要的就是变量。
变量在使用时，应始终遵循“职责单一”原则。
按作用域区分，变量可分为全局变量和局部变量。

全局变量

尽量不用或少用全局变量。
在程序设计中，全局变量是在所有作用域都可访问的变量。通常，使用不必要的全局变量被认为是坏习惯。

使用全局变量的缺点：

- 破坏函数的独立性和可移植性，使函数对全局变量产生依赖，存在耦合；
- 降低函数的代码可读性和可维护性。当多个函数读写全局变量时，某一时刻其取值可能不是确定的，对于代码的阅读和维护不利；
- 在并发编程环境中，使用全局变量会破坏函数的可重入性，需要增加额外的同步保护处理才能确保数据安全。

如不可避免，对全局变量的读写应集中封装。

规则7.1 模块间，禁止使用全局变量作接口

全局变量是模块内部的具体实现，不推荐但允许跨文件使用，但禁止作为模块接口暴露出去。
对全局变量的使用应该尽量集中，如果本模块的数据需要对外部模块开放，应提供对应函数接口。

局部变量

规则7.2 严禁使用未经初始化的变量

这里的变量，指的是局部动态变量，并且还包括内存堆上申请的内存块。
因为他们的初始值都是不可预料的，所以禁止未经有效初始化就直接读取其值。

```
void Foo(...)
{
    int data;
    Bar(data);  // Bad: 未初始化就使用
    ...
}
```

如果有不同分支，要确保所有分支都得到初始化后才能使用：

```
void Foo(...)
{
    int data;
    if (...) {
        data = 100;
    }
    Bar(data);  // Bad: 部分分支该值未初始化
    ...
}
```

未经初始化就使用，一般静态检查工具是可以检查出来的。
如 PCLint 工具，针对上述两个例子分别会报错：

> Warning 530: Symbol 'data' (line ...) not initialized Warning 644: Variable 'data' (line ...) may not have been initialized

规则7.3 禁止无效、冗余的变量初始化

如果没有确定的初始值，而仍然进行初始化，不仅不简洁，反而不安全，可能会引入更难发现的问题。

常见的冗余初始化：

```
void *buf = NULL;   // Bad: 冗余初始化，将会被后面直接覆盖
...
buf = malloc(BUF_SIZE);
...
```

对于后续有条件赋值的变量，可以在定义时初始化成默认值

```
char *buf = NULL;  // Good: 这里用 NULL 代表默认值
if (condition) {
    buf = malloc(MEM_SIZE);
}
...
if (buf != NULL) {  // 判断是否申请过内存
    free(buf);
}
```

针对大数组的冗余清零，更是会影响到性能。

```
char buf[VERY_BIG_SIZE] = {0};
memset(buf, 0, sizeof(buf));    // Bad: 冗余清零
```

无效初始化，隐藏更大问题的反例：

```
void Foo(...)
{
    int data = 0;   // Bad: 习惯性的进行初始化

    UseData(data);          // 使用数据，本应该写在获取数据后面
    data = GetData(...);    // 获取数据
    ...
}
```

上例代码，如果没有赋 0 初始化，静态检查工具可以帮助发现“未经初始化就直接使用”的问题。
但因为无效初始化，“使用数据”与“获取数据”写颠倒的缺陷，不能被轻易发现。

因此，应该写简洁的代码，对变量或内存块进行正确、必要的初始化。

C99不再限制局部变量定义必须在语句之前，可以按需定义，即在靠近变量使用的地方定义变量。
这种简洁的做法，不仅将变量作用域限制更小，而且更方便阅读和维护，还能解决定义变量时不知该怎么初始化的问题。
如果编译环境支持，建议按需定义。

规则7.4 不允许使用魔鬼数字

所谓魔鬼数字即看不懂、难以理解的数字。
魔鬼数字并非一个非黑即白的概念，看不懂也有程度，需要结合代码上下文和业务相关知识来判断

例如数字 12，在不同的上下文中情况是不一样的：
`type = 12;` 就看不懂，但 `month = year * 12;` 就能看懂。
数字 0 有时候也是魔鬼数字，比如 `status = 0;` 并不能表达是什么状态。

解决途径：
对于单点使用的数字，可以增加注释说明
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。

禁止出现下列情况：
没有通过符号来解释数字含义，如 `#define ZERO 0`
符号命名限制了其取值，如 `#define XX_TIMER_INTERVAL_300MS 300`

8 编程实践

表达式

建议8.1 表达式的比较，应当遵循左侧倾向于变化、右侧倾向于不变的原则

当变量与常量比较时，如果常量放左边，如 `if (MAX == v)` 不符合阅读习惯，而 `if (MAX > v)` 更是难于理解。
应当按人的正常阅读、表达习惯，将常量放右边。写成如下方式：

```
if (v == MAX) ...
if (v < MAX) ...
```

也有特殊情况，如：`if (MIN < v && v < MAX)` 用来描述区间时，前半段是常量在左的。

不用担心将 '==' 误写成 '='，因为 `if (v = MAX)` 会有编译告警，其他静态检查工具也会报错。让工具去解决笔误问题，代码要符合可读性第一。

规则8.1 含有变量自增或自减运算的表达式中禁止再次引用该变量

含有变量自增或自减运算的表达式中，如果再引用该变量，其结果在C标准中未明确定义。各个编译器或者同一个编译器不同版本实现可能会不一致。
为了更好的可移植性，不应该对标准未定义的运算次序做任何假设。

注意，运算次序的问题不能使用括号来解决，因为这不是优先级的问题。

示例：

```
x = b[i] + i++; // Bad: b[i]运算跟 i++，先后顺序并不明确。
```

正确的写法是将自增或自减运算单独放一行：

```
x = b[i] + i;
i++;            // Good: 单独一行
```

函数参数：

```
Func(i++, i);   // Bad: 传递第2个参数时，不确定自增运算有没有发生
```

正确的写法：

```
i++;            // Good: 单独一行
x = Func(i, i);
```

建议8.2 用括号明确表达式的操作顺序，避免过分依赖默认优先级

使用括号强调所使用的操作符，防止因默认的优先级与设计思想不符而导致程序出错；同时使得代码更为清晰可读，然而过多的括号会分散代码使其降低了可读性。下面是如何使用括号的建议。

- 一元操作符，不需要使用括号

```
x = ~a;         /* 一元操作符，不需要括号*/
x = -a;         /* 一元操作符，不需要括号*/
```

- 二元以上操作符，如果涉及多种操作符，则应该使用括号

```
x = a + b + c;          /* 操作符相同，不需要括号*/
x = Foo(a + b, c)       /* 操作符相同，不需要括号*/
if (a && b && c)        /* 操作符相同，不需要括号*/
x = (a == b) ? a : (a - b);     /* 操作符不同，需要括号*/
```

语句

规则8.2 赋值语句不要用作函数参数，不要用在产生布尔值的表达式里

作为函数参数来使用，其结果可能非预期，而且可读性差。 例：

```
int Foo(...)
{
    int var;
    var = 1;
    (void)printf("set 1st: %d, add 2nd: %d\n", var = 10, var++);    // Bad.
    var = 1;
    (void)printf("add 1st: %d, set 2nd: %d\n", var++, var = 10);    // Bad.
    ...
}
```

在if语句中，会根据条件依次判断，如果前一个条件已经可以判定整个条件，则后续条件语句不会再运行，所以可能导致期望的部分赋值没有得到运行。
例：

```
int Foo(...)
{
    int a = 0;
    int b;
    if ((a == 0) || ((b = Fun1()) > 10)) {  // Bad.
        (void)printf("a: %d\n", a);
    }
    (void)printf("b: %d\n", b);
}
```

如果布尔值表达式需要赋值操作，那么赋值操作必须在操作数之外分别进行。这可以帮助避免 '=' 和 '==' 的混淆，帮助我们静态地检查错误。
例：

```
x = y;
if (x != 0) {   // Good.
    Foo();
}
```

不能写成：

```
if ((x = y ) != 0) {    // Bad.
    Foo();
}
```

或者更坏的：

```
if (x = y) {    // Bad.
    Foo();
}
```

规则8.3 switch语句要有default分支

大部分情况下，switch语句中要有default分支，保证在遗漏case标签处理时能够有一个缺省的处理行为。

特例：
如果switch条件变量是枚举类型，并且 case 分支覆盖了所有取值，则加上default分支处理有些多余。
现代编译器都具备检查是否在switch语句中遗漏了某些枚举值的case分支的能力，会有相应的warning提示。

```
enum Color {
    RED,
    BLUE
};

// 因为switch条件变量是枚举值，这里可以不用加default处理分支
switch (color) {
    case RED:
        DoRedThing();
        break;
    case BLUE:
        DoBlueThing();
        ...
        break;
}
```

建议8.3 慎用 goto 语句

goto语句会破坏程序的结构性，所以除非确实需要，最好不使用goto语句。使用时，也只允许跳转到本函数goto语句之后的语句。

goto语句通常用来实现函数单点返回。
同一个函数体内部存在大量相同的逻辑但又不方便封装成函数的情况下，譬如反复执行文件操作， 对文件操作失败以后的处理部分代码（譬如关闭文件句柄，释放动态申请的内存等等）， 一般会放在该函数体的最后部分，在需要的地方就goto到那里，这样代码反而变得清晰简洁。 实际也可以封装成函数或者封装成宏，但是这么做会让代码变得没那么直接明了。

示例：

```
// Good: 使用 goto 实现单点返回
int SomeInitFunc(void)
{
    void *p1;
    void *p2 = NULL;
    void *p3 = NULL;

    p1 = malloc(MEM_LEN);
    if (p1 == NULL) {
```

```
        goto EXIT;
    }

    p2 = malloc(MEM_LEN);
    if (p2 == NULL) {
        goto EXIT;
    }

    p3 = malloc(MEM_LEN);
    if (p3 == NULL) {
        goto EXIT;
    }

    DoSomething(p1, p2, p3);
    return 0;   // OK.

EXIT:
    if (p3 != NULL) {
        free(p3);
    }
    if (p2 != NULL) {
        free(p2);
    }
    if (p1 != NULL) {
        free(p1);
    }
    return -1;  // Failed!
}
```

类型转换

建议8.4 尽量减少没有必要的数据类型默认转换与强制转换

当进行数据类型强制转换时，其数据的意义、转换后的取值等都有可能发生变化，而这些细节若考虑不周，就很有可能留下隐患。

如下赋值，多数编译器不产生告警，但值的含义还是稍有变化。

```
char ch;
unsigned short int exam;

ch = -1;
exam = ch; // Bad: 编译器不产生告警，此时exam为0xFFFF。
```

文件

规则8.4 避免文件过长，文件不超过2000行（非空非注释行）

文件包括源文件和头文件。
过长的文件往往意味着文件（模块）功能不单一，过于复杂。
建议根据职责进行横向拆分，或根据层次进行纵向分层。

不需人工维护的文件，则可以例外。如工具自动生成的文件。

附录

FAQ

FAQ链接：
http://rnd-isourceb.huawei.com/coding-guide/c-coding-style-guide/tree/master/faq.md

如果您有关于规范的其他问题或建议，请移步：
http://rnd-isourceb.huawei.com/coding-guide/c-coding-style-guide

相关文档

- **华为C&C++语言安全编程规范**: http://rnd-isourceb.huawei.com/coding-guide/c-cpp-secure-coding-standard
- **华为软件构建规范**: http://rnd-isourceb.huawei.com/build/build-guide

参考

- **Google Style Guides**: http://google.github.io/styleguide
- **Linux kernel coding Style**: https://www.kernel.org/doc/html/v4.18/process/coding-style.html
- **微软编程规范**: 链接
- **The C Programming Language(K&R)**: http://www.dipmat.univpm.it/~demeio/public/the_c_programming_language_2.pdf

如果发现链接失效，请及时反馈给我们

贡献者

感谢所有参与规则制订、检视、评审的专家、同事！ 感谢所有提 issue / MR 参与贡献的同事！

版本	起草	评审	批准人	修订情况
DKBA2826-2018.10	**网络**: 陈艺彪 00223933, 杨嘉 00300318 **无线**: 孙钢 00449514 **云核**: 牛平宏 00285966 **研发能力中心**: 周代兵 00340713, 陆林 00179153, 严姣红 00247381	**无线**: 郑铭00339857, 查永清 00340598, 林戎00302555, 王兵00371872, 李亚楠 00302540, 任绘锦00300562, 贾小东00339607 **网络**: 肖华山00317303, 黄维东00265762, 申力华 00247159, 阮强胜00342606, 赵广00176778, 王新华 00234133, 徐岗00399155 **云核**: 李峰00342078, 张辉 00468027 **IT产品线**: 高三海00375783, 陈雪强00338325 **公共开发**: 张伟00178855 **能源**: 任强00188021 **软件**: 包贤德00210358, 肖腾飞00257407 **中软**: 胡科平00283522, 周思义00340208 **终端**: 王斌田00221568, 张磊 00405981 **特战队**: 范一鸣00342532 **研发能力中心**: 何妙 00266241, 宋鑫00473779, 刘进00283514, 俞科技 00258636, 孙加奉90006809, 朱迪奇00268324i **海外专家**: Fan Zhang 00407699, Eric Li 00396581, Bin Liang 00452313, Kang Xi 00385799, Yan Chen 00754567, Wenzhe Zhou 00725915, Chun Liu 00415003 **外部顾问**: Steve Tockey (Construx Corp.)	张来发 00338367	较大重构，包括: 章节调整、去除掉原则、去除掉不适合规范约束的条目，补充条目说明、补充例子等等

版本	起草	评审	批准人	修订情况
DKBA2826-2016.05	**研发能力中心**: 郭曙光 00340987	**网络**: 张伟00342388, 周灿00286455 **IP开发部**: 陈艺彪00223933 **无线**: 邱霖00340162 **电软**: 赵玉锡00232229 **研发能力中心**: 吴敏00326311, 刘进00283514, 曹锦业00246228, 陈宇00291422, 朱喜红00210657		新增规则1.9、规则2.8、规则2.9、建议2.7、建议2.8、建议2.9、规则4.4、建议4.6、建议4.7、建议4.8、规则6.6、规则6.7、规则8.9、建议8.4、建议8.5、建议8.6、规则9.7、建议11.2、建议11.2。 安全章节整体移出到C&C++安全编程规范。 单元测试章节合入可测性章节。
DKBA2826-2013.07	**研发能力中心**: 郭曙光 00121837	**网络**: 张伟00118807 **研发能力中心** 王红超00134169		修改部分规则的说明和例子。

版本	起草	评审	批准人	修订情况
DKBA2826-2011.05	**PSST质量部**: 郭曙光00121837 **网络**: 张伟00118807, 周灿00056781, 王晶00041937, 陈艺彪00036913 **IP开发部**: 薛治00038309 **核心网**: 张小林00058208, 王德喜00040674, 李明胜00042021, **软件公司**: 文滔00119601 **无线**: 刘爱华00162172 **中研**: 谭洪00162654	**PSST质量部**: 李重霄00117374, 郭永生00120218 **核心网**: 张进柏00120359 **中研**: 张建保00116237 **无线**: 苏光牛00118740, 郑铭00118617, 陶永祥00120482 **软件公司**: 周代兵00120359, 刘心红00118478, 朱文琦00172539 **网络**: 王玎00168059, 黄维东00265762 **IP开发部**: 饶远00152313		