C语言编程规范

目的

规则并不是完美的，通过禁止在特定情况下有用的特性，可能会对代码实现造成影响。但是我们制定规则的目的“为了大多数程序员可以得到更多的好处”， 如果在团队运作中认为某个规则无法遵循，希望可以共同改进该规则。 参考该规范之前，希望您具有相应的C语言基础能力，而不是通过该文档来学习C语言。

1. 了解C语言的ISO标准；
2. 熟知C语言的基本语言特性；
3. 了解C语言的标准库；

总体原则

代码需要在保证功能正确的前提下，满足**可读、可维护、安全、可靠、可测试、高效、可移植**的特征要求。

约定

规则：编程时必须遵守的约定
建议：编程时必须加以考虑的约定

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。

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

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(UpperCamelCase)**和小驼峰(lowerCamelCase)**

规则1.1 标识符命名使用驼峰风格

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
变量，函数参数，宏参数，结构体中字段，联合体中成员	小驼峰
宏，常量，枚举值，goto 标签	全大写，下划线分割

注意：
上表中`常量`是指，全局作用域下，const 修饰的基本数据类型、枚举、字符串类型的变量，不包括数组、结构体和联合体。
上表中`变量`是指除常量定义以外的其他变量，均使用小驼峰风格。

建议1.1 作用域越大，命名应越精确

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

为了命名更精确，必要时可以增加模块前缀。
模块前缀与命名主体之间，按驼峰方式连接。
示例：

```
int PrefixFuncName(void);    // OK: 驼峰方式，形式上无前缀，内容上有前缀

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

文件命名

建议1.2 文件命名统一采用小写字符

文件名命名只允许使用小写字母、数字以及下划线(_)。
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 函数的命名遵循阅读习惯

动作类函数名，可以使用动宾结构。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1.2 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

全局变量应当尽量少使用，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(void)
{
    static int pktCount = 0;
    ...
}
```

注意： 常量本质也是全局变量，但如果命名风格是全大写，下划线连接的格式，则不适用当前规则。

建议1.4 局部变量应该简短，且能够表达相关含义

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
};

typedef int (*NodeCmpFunc)(struct Node *a, struct Node *b);
```

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;
    struct tagNode *next;
} Node;              // 类型主体用大驼峰风格
```

宏、常量、枚举命名

宏、枚举值采用全大写，下划线连接的格式。
常量推荐采用全大写，下划线连接风格。作为全局变量，也可以保持与普通全局变量命名风格相同。
这里常量如前文定义，是指基本数据类型、枚举、字符串类型的全局 const 变量。

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。

宏举例：

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

```
#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
void Foo(int);
#endif
```

常量举例：

```
const int VERSION = 200;     // OK.

const enum Color DEFAULT_COLOR = BLUE; // OK.

const char PATH_SEP = '/';  // OK.

const char * const GREETINGS = "Hello, World!"; // OK.
```

非常量举例：

```
// 结构体类型，不符合常量定义
const struct MyType g_myData = { ... };      // OK: 用小驼峰

// 数组类型，不符合常量定义
const int g_xxxBaseValue[4] = { 1, 2, 4, 8 };    // OK: 用小驼峰

int Foo(...)
{
    // 局部作用域，不符合常量定义
    const int bufSize = 100;     // OK: 用小驼峰
    ...
}
```

枚举举例：

```
// 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
enum BaseColor {
    RED,
    GREEN,
    BLUE
};
```

建议1.5 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

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。
控制行宽长度可以间接的引导开发去缩短函数、变量的命名，减少嵌套的层数，提升代码可读性。
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。

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 使用 K&R 缩进风格

K&R风格
换行时，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
}
```

函数声明和定义

规则2.3 函数声明、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。

函数参数列表换行时，应合理对齐。
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 函数调用参数列表换行时保持参数进行合理对齐

函数调用时，函数参数列表如果换行，应该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，操作符放行末

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。一般在较低优先级操作符或连接符后面截断，操作符或连接符放在行末。
操作符、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，或进行合理对齐

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
    }
};
```

注意：

- 左大括号放行末时，对应的右大括号需另起一行
- 左大括号被内容跟随时，对应的右大括号也应跟随内容

规则2.11 结构体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
```

选择一种风格，并保持一致性。

选择"*"跟随类型风格时，避免一行同时声明带指针的多个变量。

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

选择"*"跟随变量风格时，可能会存在无法紧跟的情况。
无法跟随时就不跟随，不要破坏风格一致性。

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

注意，任何时候 "*" 不要紧跟 const 或 restrict 关键字。

编译预处理

规则2.12 编译预处理的"#"默认放在行首，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。

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
- 二元操作符（= + - < > * / % | & ^ <= >= == !=）左右两侧加空格
- 一元操作符（& * + - ~ !）之后不要加空格
- 三目操作符（? :）符号两侧均需要空格
- 结构体中表示位域的冒号，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
- 大括号内部两侧有无空格，左右必须保持一致
- 逗号、分号、冒号（不含三目操作符和表示位域的冒号）紧跟前面内容无空格，其后需要空格
- 函数参数列表的小括号与函数名之间无空格
- 类型强制转换的小括号与被转换对象之间无空格
- 数组的中括号与数组名之间无空格
- 涉及到换行时，行末的空格可以省去

对于大括号内部两侧的空格，**建议**如下：

- 一般的，大括号内部两侧建议加空格
- 对于空的，或单个标识符，或单个字面常量，空格不是必须 如：'{}', '{0}', '{NULL}', '{"hi"}' 等
- 连续嵌套的多重括号之间，空格不是必须 如：'{{0}}', '{{ 1, 2 }}' 等 错误示例：'{ 0, {1}}'，不属于连续嵌套场景，而且最外侧大括号左右不一致

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建议加空格
```

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2.4 合理安排空行，保持代码紧凑

减少不必要的空行，可以显示更多的代码，方便代码阅读。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；
- 函数内部、类型定义内部、宏内部、初始化表达式内部，不使用连续空行
- 不使用连续 **3** 个空行，或更多
- 大括号内的代码块首行之前和末行之后不要加空行。

```
ret = DoSomething();

if (ret != OK) {     // Bad：返回值判断应该紧跟函数调用
    return -1;
}
```

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



int Bar(void)        // Bad：最多使用连续2个空行。
{
    ...
}
```

```
int Foo(void)
{

    DoSomething();  // Bad：大括号内部首尾，不需要空行
    ...

}
```

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。

使用英文进行注释。

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

/*

- Copyright (c) 2020 XXX
- Licensed under the Apache License, Version 2.0 (the "License");
- you may not use this file except in compliance with the License.
- You may obtain a copy of the License at
-
- `http://www.apache.org/licenses/LICENSE-2.0`
-
- Unless required by applicable law or agreed to in writing, software
- distributed under the License is distributed on an "AS IS" BASIS,
- WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
- See the License for the specific language governing permissions and
- limitations under the License.
- /

函数头注释

规则3.2 禁止空有格式的函数头注释

并不是所有的函数都需要函数头注释；
函数原型无法表达的信息，加函数头注释辅助说明;

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 case语句块结束时如果不加break/return，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或return，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;
}
```

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。

建议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（并不一定同名），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(void);  // Good：头文件中声明对外接口

#endif
```

foo.c 内容

```
static void Bar(void);  // 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(void)
{
    Bar();
}

static void Bar(void)
{
    // 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4.2 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，具体见建议4.1。

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4.1 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，可通过架构优化来避免。

规则4.2 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；建议考虑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 timer 模块的 timer.h，其目录为 `timer/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 TIMER_INCLUDE_TIMER_H
#define TIMER_INCLUDE_TIMER_H

...

#endif
```

规则4.3 禁止通过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(void);   // Bad: 通过 extern 的方式引用外部函数

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

b.h 内容

```
int Foo(void);
```

b.c内容

```
int Foo(void)
{
    // 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4.4 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(void);

#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(void);
}
```

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。

---

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：
考虑代码的聚合性与功能的全面性，某些函数可能会超过50行，但前提是不影响代码的可读性与简洁。
这些例外的函数应该是极少的，例如特定算法处理。

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：
原代码嵌套层数是 3：

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

使用`卫语句`重构，嵌套层数变成 2：

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

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

    return DealMsg(..);
}
```

例外：
考虑代码的聚合性与功能的全面性，某些函数嵌套可能会超过4层，但前提是不影响代码的可读性与简洁。
这些例外的函数应该是极少的。

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。

函数参数

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 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5.4 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5.5 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
```

注意： 指针参数要不要加 const 取决于函数设计，而不是看函数实体内有没有发生“修改对象”的动作。

建议5.6 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

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5.7 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(void);
```

inline.c

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

other.c

```
#include "inline.h"
int OtherFunc(void)
{
    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 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(void)
{
    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操作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来解决，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，单独的数字或标识符加括号毫无意义。

```
#define SOME_CONST       100     // Good: 单独的数字无需括号
#define ANOTHER_CONST    (-1)    // Good: 负数需要使用括号

#define THE_CONST SOME_CONST     // Good: 单独的标识符无需括号
```

下列情况需要注意：

- 宏参数参与 '#', '##' 操作时，不要加括号
- 宏参数参与字符串拼接时，不要加括号
- 宏参数作为独立部分，在赋值（包括+=, -=等）操作的某一边时，无需括号
- 宏参数作为独立部分，在逗号表达式，函数或宏调用列表中，无需括号

举例：

```
#define MAKE_STR(x) #x                  // x 不要加括号

#define HELLO_STR(obj) "Hello, " obj    // obj 不要加括号

#define ADD_3(sum, a, b, c) (sum = (a) + (b) + (c)) // a, b, c 需要括号；而 sum 无需括号

#define FOO(a, b) Bar((a) + 1, b)       // a 需要括号；而 b 无需括号
```

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

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)
```

例外：

- 包含 break, continue 语句的宏可以例外。使用此类宏务必特别小心。
- 宏中包含不完整语句时，可以例外。比如用宏封装 for 循环的条件部分。
- 非多条语句，或单个 if/for/while/switch 语句，可以例外。

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，如果参数包含函数调用，宏展开后，函数可能会被重复调用。
如果函数执行结果相同，则存在浪费；如果函数多次调用结果不一样，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.
```

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: 内存泄漏
```

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。

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；
对于返回值判断等异常处理场景可以例外。

注意： **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。**

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：

```
int cnt = 0;    // Bad: 冗余初始化，将会被后面直接覆盖
...
cnt = GetXxxCnt();
...
```

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。

例外：
遵从“安全规范”要求，指针变量、表示资源描述符的变量、BOOL变量不作要求。

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

可以使用括号强调表达式操作顺序，防止因默认的优先级与设计思想不符而导致程序出错。
然而过多的括号会分散代码使其降低了可读性，应适度使用。

当表达式包含不常用，优先级易混淆的操作符时，推荐使用括号，比如位操作符：

```
c = (a & 0xFF) + b;      /* 涉及位操作符，需要括号 */
```

语句

规则8.2 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。
```