C语言编程指南 V1.0

目录

前言

由于C语言是在计算机运行速度比较低与内存空间比较小的1970年代初期研制的，那时的操作系统、编译系统与数据库管理系统等系统软件都是用机器语言或汇编语言编写的，C语言的研制者为了能够用C编写UNIX操作系统，不得不把效率（执行时间与适应内存）放在第一位。因此，C语言的设计目标是提供一种能够以简易的方式编译、处理低级存储器、生成尽可能少的机器代码以及不需要任何运行环境支持便能运行的程序设计语言。

C语言描述问题比汇编语言简单、所需工作量少、易于调试修改与移植，而代码质量与汇编语言代码相当（C程序生成的代码一般只比汇编语言代码效率低10~20%）。从而，与同时代语言及目前的许多语言相比，C语言在可读性、可维护性、可移植性、安全性和可靠性等方面有着先天的不足，站在现在的角度，C语言的数组、指针等机制也给编写高安全的程序带来了很大的麻烦。而可读性、可维护性、可移植性、安全性与可靠性恰恰是当今评价软件质量的重要指标。

本指南为提高代码的可读性、可维护性、可移植性、安全性与可靠性，提供编程规范条款，力争系统化、易使用、易检查。

本指南条款分为原则和规则两个类别，通过标题前的编号标识。标识'P'为原则（单词Principle首字母），标识'G'为规则（单词Guideline的首字母）。原则的编号方式为`P.Number`，规则的编号方式为`G.Element.Number`。Number是从01开始递增的两位阿拉伯数字，Element为领域知识中关键元素的3位英文字母缩略语。

Element	解释	Element	解释
NAM	命名	CMT	注释
FMT	格式	PRE	预处理
INC	头文件	TYP	数据类型
CNS	常量	VAR	变量
EXP	表达式	CTL	控制语句
DCL	声明与初始化	INT	整数
ARR	指针和数组	STR	字符串
AST	断言	FUD	函数设计
FUU	函数使用	MEM	内存
FIL	文件	OTH	其他

1 代码风格

代码风格一般包含标识符的命名风格、排版与格式风格，注释风格。一致的编码习惯与风格，会使代码更容易阅读、理解更容易维护。

1.1 命名

P.01 标识符命名应符合阅读习惯

【描述】
标识符的命名要清晰、明了，有明确含义，容易理解。符合英文阅读习惯的命名将明显提高代码可读性。

一些好的实践包括但不限于：

- 使用正确的英文单词并符合英文语法，不要使用拼音
- 仅使用常见或领域内通用的单词缩写
- 布尔型变量或函数避免使用否定形式

P.02 作用域越大，命名越精确；反之应简短

【描述】
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是否会冲突。
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);               // 不符合：描述不精确
int GetActiveConnectCount(void); // 符合
```

对于函数局部变量，或者结构体、联合体中的成员变量，其命名在能够准确表达含义的前提下应该尽量简短，避免冗余信息的重复描述。

【反例】

```
void Func(void)
{
    enum PowerBoardStatus powerBoardStatusOfSlot; // 不符合：局部变量有点长
    ...
    powerBoardStatusOfSlot = GetPowerBoardStatus(slot);
    if (powerBoardStatusOfSlot == POWER_OFF) {
        ...
    }
    ...
}
```

【正例】

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

某些场景，甚至可以使用单字符命名的变量，只要无二义性即可，例：

```
int i; // 符合：用单字符命名循环变量
for (i = 0; i < COUNTER_RANGE; i++) {
    ...
}
```

```
int MyMul(int a, int b) // 符合：能够准确表达含义
{
    return a * b;
}
```

G.NAM.01 使用统一的命名风格

【描述】
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)**

内核风格(unix_like)
又称蛇形风格(snake_case)。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'

应使用统一的命名风格，当前条款推荐“驼峰风格”，具体规则如下：

类别	命名风格	形式
函数，结构体类型，枚举类型，联合体类型，typedef 定义的类型	大驼峰，或带模块前缀的大驼峰	AaaBbb, XXX_AaaBbb
局部变量，函数参数，宏参数，结构体中字段，联合体中成员	小驼峰	aaaBbb
全局变量（在函数外部定义的变量）	带 'g_' 前缀的小驼峰	g_aaaBbb
宏（不包括函数式宏），枚举值，goto 标签	全大写下划线分割	AAA_BBB
函数式宏	全大写下划线分割，或大驼峰，或带模块前缀的大驼峰	AAA_BBB, AaaBbb, XXX_AaaBbb
常量 （在函数外部定义由const修饰的基本数据类型、枚举类型、字符串类型）	全大写下划线分割	AAA_BBB

关于“模块前缀”：

- 仅大驼峰命名风格的符号，可选加模块前缀。
- 应只选用一种前缀形式，保持风格统一（仅选用XXX_AaaBbb，或仅选用XxxAaaBbb）。
- 模块前缀尽量简短且不超过**2**级（XXX_YYY_AaaBbb, XxxYyyAaaBbb）。

关于“函数式宏”：

- 函数式宏的命名风格优先与宏一样，采用“全大写下划线分割”。
- 特殊情况，允许将函数式宏的命名风格与函数一样，但这种情况应该是极少的。

【正例】

```
int MyCmp(int a, int b); // 符合：函数大驼峰，参数小驼峰（单字符变量也符合小驼峰定义）

enum MyColor {                  // 符合：枚举类型，大驼峰
    BLACK,                      // 符合：枚举值，全大写，下划线分割
    WHITE
} g_bgColor = WHITE;            // 符合：全局变量，带 g_前缀的小驼峰

int XXX_YYY_FuncName(void);     // 符合：函数，两级前缀，大模块加小模块

const int NAME_MAX_LEN = 100; // 符合：符合上述常量定义，用全大写下划线分割

/* “单词缩写”应当作单个单词处理，以提高可读性，如：
 * 对包含HTTP（HyperText Transfer Protocol）缩写的函数命名
 */
int GetActiveHttpClientCnt(void);
```

1.2 注释

P.03 注释跟代码一样重要，应按需注释

【描述】
一般的，尽量通过清晰的软件架构，良好的符号命名来提高代码可读性；然后在需要的时候，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、无歧义，信息全面且不冗余。
需要注释的地方没有注释，代码则难以被读懂；而包含无用、重复信息的冗余注释不仅浪费维护成本，还会弱化真正有用的注释，最终让所有注释都不可信。

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。

使用流利的中文或英文进行注释。
为降低沟通成本，应使用团队内最擅长、沟通效率最高的语言写注释；注释语言由开发团队统一决定。

G.CMT.01 注释符与注释内容间要有1空格

【描述】
注释符使用 `/*` `*/` 和 `//` 都是可以的。
注释符与注释内容之间要有1空格。

使用如下的单行、多行注释风格：

```
// 单行注释
```

```
/* 另外一种单行注释 */
```

```
/*
 * 多行注释
 * 第二行
 */
```

```
// 另外一种多行注释
// 第二行
```

【注意】

- 若产品选用如 doxygen 来基于注释生成代码文档，则应由产品制定统一的各类广义的注释符及注释风格。
- “注释内容”有可能也包含空格，比如：

```
/*
 * 这是一段注释内容也包含空格的例子。
 * XX参数解释如下：
 *      a - 参数 a 说明
 *      b - 参数 b 说明
 */
```

G.CMT.02 代码注释置于对应代码的上方或右边

【描述】
针对代码的注释，应该置于对应代码的上方或右方。

代码上方的注释，与代码行间无空行，保持与代码一样的缩进。
例：

```
// 这是 Foo() 的注释
int Foo(void)
{
    // 这是变量 a 的注释
    int a = INIT_A_VALUE;
    ...
}
```

代码右边的注释，与代码之间，至少留1空格。
例：

```
int foo = 100;  // 这里是注释内容，与代码至少留1空格
```

右置格式在适当的时候，上下对齐会更美观。
例：

```
#define A_CONST 100          // 此处两行注释属于同类
#define ANOTHER_CONST 200    // 可保持左侧对齐
```

通常右置注释内容不宜过多；当右置注释超过行宽时，请考虑将注释置于代码上方。

G.CMT.03 文件头注释包含版权说明

【描述】
文件头注释应首先包含版权说明。
如果文件头注释需要增加其他内容，可以后面补充。
比如：文件功能说明，作者、创建日期、注意事项等等。

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

关于版本说明，应注意：

- 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，如特性扩展，重大重构等。
- 可以只写一个创建年份，后续文件修改则不用更新版权声明。
  如：`版权所有 (c) XXX技术有限公司 2018`

文件头注释举例：

```
/*
 * 版权所有 (c) XXX技术有限公司 2012-2018
 * XX 功能实现
 * 注意：
 *       - 注意点 1
 *       - 注意点 2
 */
```

编写文件头注释应注意：

- 文件头注释应该从文件顶头开始。如果包含“关键资产说明”类注释，则应紧随其后。
- 保持统一格式。具体格式由项目或更大范围统一制定。格式可参考上面举例。
- 保持版面工整，若内容过长，超出行宽要求，换行时应注意对齐。对齐可参考上述例子 '注意点'。
- 首先包含“版权许可”，然后包含其他可先选内容。其他选项按需添加，并保持格式统一。
- 不要空有格式，无内容。如上述例子，如果 '注意：' 后面无内容，不应编写。

G.CMT.04 不写空有格式的函数头注释

【描述】
要像写代码注释一样**按需**去写函数头注释。

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，却又希望读者知道的信息，才需要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。

【正例】

```
/*
 * 返回实际写入的字节数，-1表示写入失败
 * 注意，len小于INT_MAX
 */
int WriteData(const unsigned char *buf, size_t len);
```

【反例】
下面注释示例中的问题：

- 参数、返回值，空有格式没内容
- 注释中的函数名信息冗余

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

1.3 格式

1.3.1 编码

G.FMT.01 非纯ASCII码源文件使用 UTF-8 编码

【描述】
对于非纯 ASCII 源文件，使用 UTF-8 格式。
请正确配置你的编辑器。

1.3.2 缩进

G.FMT.02 使用空格进行缩进，每次缩进4个空格

【描述】
使用空格而不是制表符('\t')进行缩进，每次缩进为 **4** 个空格。
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。

1.3.3 大括号

G.FMT.03 使用统一的大括号换行风格

【描述】
典型的大括号换行风格包括**K&R风格**和**Allman风格**，本指南中示例通常使用K&R的大括号风格。

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

int Foo(int a)
{               // 函数左大括号独占一行，放行首
    if (a > 0) {
        ...
    } else {    // 右大括号、"else"、以及后续的左大括号均在同一行
        ...
    }           // 右大括号独占一行
    ...
}
```

Allman风格
换行时，左大括另起并独占一行，保持与上一行相同缩进；
右大括号独占一行，除非后面跟着 do 语句中的 while，或者逗号、分号。
如：

```
struct MyType
{               // 另起并独占一行
    ...
};              // 右大括号后面紧跟分号

int Foo(int a)
{
    if (a > 0)
    {
        ...
    }
    else        // 前后的左右大括号均独占一行，所以 'else' 也只能独占一行
    {
        ...
    }
    ...
}
```

1.3.4 行宽

G.FMT.04 一行只有一条语句

【描述】
一行只写一条语句。
例：

```
int Foo(void)
{
    int a = 10; // 符合.

    for (int i = 0; i < CNT; i++) { // 符合: 这里有分号,但并不是语句
        Bar();  // 符合.
    }

    if (cond) { Func1(); } else { Func2(); } // 不符合：一行包含两个复合语句

    ...
}
```

G.FMT.05 行宽不超过 120 个字符

【描述】
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。
建议每行字符数不要超过 **120** 个；除非超过 **120** 能显著增加可读性，且不会隐藏信息。

【例外】
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
```

G.FMT.06 换行时将操作符留在行末，新行缩进一层或进行同类对齐

【描述】
当语句过长，或者换行后有更好的可读性时，应根据层次或操作符优级先择合适的断行点进行换行，并将表示未结束的操作符或连接符号留在行末。
操作符、连接符放在行末，表示“未结束，后续还有”。
新行缩进一层，或者保持同类对齐。

长表达式举例：

```
// 假设下面第一行已经不满足行宽要求
if (currentValue > MIN &&  // 符合：换行后，布尔操作符放在行末
    currentValue < MAX) {  // 符合: 与(&&)操作符的两个操作数同类对齐
    DoSomething();
    ...
}

flashPara.flashEndAddr = flashPara.flashBaseAddr +  // 符合: 加号留在行末
                         flashPara.flashSize;       // 符合: 加法两个操作数对齐
```

函数参数列表举例：

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

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


ReturnType result = FunctionName(paramName1, paramName2,
    paramName3, paramName4, paramName5);     // 符合：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。

```
// 符合：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 = DealWithStructLikeParams(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
```

初始化语句举例：

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

```
// 符合：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 }, // 符合
    { 3, 2, 3, 4 }, { 4, 2, 3, 4 }
};

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

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

对于初始化语句中的大括号，遵循：

- 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

1.3.5 函数

G.FMT.07 函数的返回类型及修饰符与函数名同行

【描述】
声明定义函数时，函数的返回值类型以及其他修饰符，保持与函数名同一行。

例：

```
static inline int ShortFunc(int a, int b);  // 符合

// 符合：参数可以与函数名不同行
static inline int LongFuncName(int longParamName1,
                               int longParamName2,
                               int longParamName3,
                               int longParamName4);
```

1.3.6 语句

G.FMT.08 条件、循环语句使用大括号

【描述】
包括 if/for/while/do-while 语句应使用大括号，即复合语句。

理由：

- 代码逻辑直观，易读；
- 在已有代码上增加新代码时不容易出错；
- 对于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）。

【正例】

```
// 示例1
if (objectIsNotExist) { // 符合：单行条件语句也加大括号
    return CreateNewObject();
}

// 示例2
for (int i = 0; i < someRange; i++) { // 符合：使用了大括号
    DoSomething();
}

// 示例3
while (condition) {} // 符合：即使循环体是空，也应使用大括号

// 示例4
while (condition) {
    continue;        // 符合：continue 表示空逻辑，使用大括号
}
```

【反例】

```
// 示例1
for (int i = 0; i < someRange; i++)
    DoSomething();  // 不符合：应该加上括号


// 示例2
while (condition);  // 不符合：很容易让人误解循环体是 DoSomething() 调用
DoSomething();
```

G.FMT.09 case/default 语句相对 switch 缩进一层

【反例】

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

【正例】

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

G.FMT.10 指针类型"*"跟随变量或者函数名

【描述】
声明或定义指针变量或者返回指针类型函数时，"*" 应该靠右跟随。
例：

```
int *p1;  // 符合
int* p2;  // 不符合
int*p3;   // 不符合：两边都没空格
int * p4; // 不符合：两边都有空格
```

```
struct Foo *CreateFoo(void); // 符合："*"跟随函数名
```

下列情况需要特别注意：

- 当"*"与变量或函数名之间有其他修饰符，无法跟随时，此时也不要跟随修饰符

```
char * const VERSION = "V100";     // 符合：当有 const 修饰符时，"*"两边都有空格
int Foo(const char * restrict p); // 符合：当有 restrict 修饰符时，"*"两边都有空格
```

任何时候 "*" 不要紧跟 const 或 restrict 关键字。

- 当右侧没有变量或函数名时，"*" 可以跟随类型

```
sz = sizeof(int*); // 符合：右侧没有变量，"*"跟随类型
```

1.3.7 空格与空行

G.FMT.11 用空格突出关键字和重要信息

【描述】
空格应该突出关键字和重要信息。总体建议如下：

- 行末不要加空格
- 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
- 函数或宏参数列表的小括号与函数名或宏名之间无空格
- 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;                // 符合：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}; // 符合：数组初始化时，大括号内空格可选
int arr[] = { 10, 20 };  // 符合：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);
                      ^          // 不符合：逗号后面应该有空格

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

指针和取地址

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

操作符：

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

if (x && !y)        // 符合：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; // 符合：二元操作符前后要加空格
v = w * (x + z);   // 符合：括号内的表达式前后不需要加空格
```

循环和条件语句：

```
if (condition) { // 符合：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 {         // 符合：else关键字和大括号之间加空格
    ...
}

// 符合：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
while (condition) {}

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

switch (var) { // 符合：switch 关键字后面有1空格
    case 0:    // 符合：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。

G.FMT.12 避免连续3个或更多空行

【描述】
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
- 大括号内的代码块行首之前和行尾之后不要加空行。

```
ret = DoSomething();

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

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



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

```
int Foo(void)
{

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

}
```

2 编程实践

2.1 预处理

2.1.1 宏

G.PRE.01 使用函数代替函数式宏

【描述】
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。参见示例代码。
- 宏展开时宏参数不求值，可能会产生非预期结果。参见规则定义宏时，要使用完备的括号和规则禁止把带副作用的表达式作为参数传递给函数式宏。
- 宏没有独立的作用域，跟控制流语句配合时，可能会产生如规则包含多条语句的函数式宏的实现语句必须放在 do-while(0) 中描述的非预期结果。
- 宏的技巧性太强，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;
}

void 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));
}
```

上面的示例代码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/函数的参数求值只会进行一次。
- 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。

对于性能要求高的产品代码，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__)
```

G.PRE.02 定义宏时，要使用完备的括号

【描述】
宏展开时只做文本替换，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。

比如：

```
#define C_LEN A_LEN + B_LEN     // 不符合.
```

上述宏在展开时，A_LEN 与 B_LEN 的加法并不一定是优先计算。
正确的写法应该是：

```
#define C_LEN (A_LEN + B_LEN)   // 符合.
```

带参数的宏更容易出现问题，比如：

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

下面这样调用该宏，执行结果跟预期不符：
`100 / SUM(2, 8)` 将扩展成 `(100 / 2) + 8`，而预期结果是 `100 / (2 + 8)`。

这个问题的解决方法如下所示：

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

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         // 符合：单独的数字无需括号
#define ANOTHER_CONST   (-1)    // 符合：负数需要使用括号
#define THE_CONST   SOME_CONST  // 符合：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，可以不加括号

- 宏参数作为独立部分，在逗号表达式，函数或宏调用列表中，可以不加括号

举例如下：

```
// x 不要加括号
#define MAKE_STR(x) #x

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

// a, b 需要括号；而 value 可以不加括号
#define UPDATE_VALUE(value, a, b) (value = (a) + (b))

// a 需要括号；而 b 可以不加括号
#define FOO(a, b) Bar((a) + 1, b)
```

G.PRE.03 包含多条语句的函数式宏的实现语句必须放在 do-while(0) 中

【描述】
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，示例代码稍不符本条款）：

```
// 不符合
#define FOO(x) \
    (void)printf("arg is %d\n", (x)); \
    DoSomething((x));
```

当像如下示例代码这样调用宏FOO，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 < MAX_TIMES; 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)); \
}
```

但是，如下示例代码，会出现编译报错(else没有与之匹配的if语句)：

```
if (condition)
    FOO(MAX_MONTH);
else
    FOO(MAX_YEAR);
```

更好的写法是用 do-while(0) 把宏FOO执行体括起来，如下所示：

```
// 符合
#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(参见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。

G.PRE.04 禁止把带副作用的表达式作为参数传递给函数式宏

【描述】
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。

【反例】
如下所示，宏SQUARE本身没有问题，但是使用时将带副作用的表达式a++传入导致a的值在SQUARE执行后的结果跟预期不符：

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

int a = 5;
int b = SQUARE(a++); // 不符合： 展开后表达式中有2个 "a++"，其结果可能是非预期的。
```

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。

【正例】
代码做如下修改：

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

G.PRE.05 函数式宏定义中慎用 return、goto、continue、break 等改变程序流程的语句

【描述】
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，存在过度封装，容易导致资源泄漏等问题。

【反例】
如下是宏封装 return 容易导致过度封装和使用的场景：
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!"); // 符合

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

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

如下是宏封装 return 也容易引发内存泄漏的场景：

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

...

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

mem2 = MemAlloc(STR_SIZE_MAX);
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；

【例外】
用于返回值判断等异常处理场景的宏可以包含改变程序流程的语句。
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。
比如：

```
#define RETURN_IF(condition, retValue) \
    if (condition) { \
        RecordFailInfo(__FILE__, __LINE__); \
        return retValue; \
    }
```

G.PRE.06 函数式宏要简短

【描述】
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膨胀。建议函数式宏不要超过10行（非空非注释）。

G.PRE.07 宏的名称不应与关键字相同

【描述】
如果使用宏改变了C语言关键字的含义，在修改或维护代码时，可能会因为不能准确理解宏的语义，导致错误使用该宏。

G.PRE.08 禁止宏调用参数中出现预编译指令

【描述】
宏参数如果包括预编译指令会导致程序产生未定义行为。

【反例】
如下代码可能会导致程序出现未定义行为。

```
// 宏定义
#define MACRO_NAME(msg)  ...

// 宏调用
MACRO_NAME(
#ifdef PLATFORM1
    msg1
#elif PLATFORM2
    msg2
#else
    msg3
#endif
);
```

【正例】

```
// 宏定义
#define MACRO_NAME(msg)  ...

// 宏调用
#ifdef PLATFORM1
    MACRO_NAME(msg1);
#elif PLATFORM2
    MACRO_NAME(msg2);
#else
    MACRO_NAME(msg3);
#endif
```

G.PRE.09 宏定义不应以分号结尾

【描述】
当宏定义的结尾有分号时，从代码中难以直观发现语句的结束，降低了程序语句的可读性，并增加了程序员在调用宏时额外增加分号的可能（额外增加的分号可能导致程序语法错误或流程错误），因此，宏末尾的分号应由调用者提供。

【反例】

```
#define ERR_CODE     10;     // 不符合：尾部多了一个分号

printf("CODE: %d", ERR_CODE);    // 语法错误。
```

【正例】

```
#define ERR_CODE     10     // 符合：去掉尾部的分号

printf("CODE: %d", ERR_CODE);   // 语法正确。
```

G.PRE.10 宏定义不应依赖宏外部的局部变量名

【描述】
宏定义中要使用的变量都要求作为参数传递给宏。
如果宏定义中直接使用宏外部的局部变量名，会导致宏的可重用性差，而且不利于理解。

【反例】

```
#define INIT(x) do { \
    count = (x)->y->length; \
} while (0)

...
int count;
INIT(msg);
```

【正例】

```
#define INIT(x) ((x)->y->length)

...
int count = INIT(msg);
```

2.1.2 条件编译

G.PRE.11 #if或#elif预处理指令中的常量表达式的值应为布尔值

【描述】
`#if ... #endif`或`#if ... #elif ... #endif`的预处理指令中常量表达式的值应为布尔值。
如果预处理指令中的常量表达式仅是一个开关用途的宏，可以使用数值0或1作为该宏的值。

【反例】

```
#define VERSION 3320

#if 3320    // 不符合
...
#endif

#if 0       // 不符合
...
#endif

#if 1       // 不符合
...
#endif

#if VERSION // 不符合
...
```

```
#endif
```

【正例】

```
#define OLD_VERSION 1000
#define VERSION 3320
#define ENABLE_MMX   1       // 定义开关用途的宏
#define DISABLE_FLOAT 0      // 定义开关用途的宏


#if ENABLE_MMX             // 符合
...
#endif

#if DISABLE_FLOAT           // 符合
...
#endif

#if defined(VERSION)       // 符合
...
#endif

#if VERSION > OLD_VERSION // 符合
...
#endif
```

G.PRE.12 #if或#elif预处理指令中的常量表达式被求值前应确保其使用的标识符是有效的

【描述】

在宏扩展之后，如果预处理指令中的常量表达式含有编译器不识别的标识符，则会将这些标识符替换为常量0，存在这种标识符通常认为是编程错误。

因此，良好的做法是在预处理指令中的常量表达式被求值前，使用`#if defined`、`#ifdef`等预处理指令检查标识符，确保是有效的。

【反例】

```
#if VERSION == 4000 // 不符合：VERSION 可能未定义
...
#endif
```

【正例】

```
#ifdef VERSION        // 符合
#if VERSION == 4000
...
#endif
#elif
...
#endif
```

G.PRE.13 所有#else、#elif、#endif和与之对应的#if、#ifdef、#ifndef预处理指令应出现在同一文件中

【描述】
当使用多个文件来构成这些预处理块时，难以直观发现代码块的关联，增加了阅读和维护的复杂度，更容易产生错误。

【反例】
如下代码示例中，未在相同文件中闭合预处理块：

```
// sample.h 开始
#if defined(__VXWORKS__)  // 未在相同文件中闭合该预处理块
#include "vx_config.h"


// 这里结束前，本应当有 #endif 与前面的 #if defined 形成闭合
// sample.h 结束
```

【正例】
如下代码示例中，在相同文件中闭合预处理块：

```
// sample.h 开始
#if defined(__VXWORKS__)    // 在相同文件中闭合该预处理块
#include "vx_config.h"
#endif
// sample.h 结束
```

2.2 头文件

G.INC.01 在头文件中声明需要对外公开的接口

【描述】
通常情况下，每个.c文件都应有一个相应的.h文件（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。

【正例】
以下代码中函数Foo是对外提供的接口，函数Bar是内部函数，不对外提供。

foo.h 内容

```
#ifndef FOO_H
#define FOO_H

void Foo(void); // 符合：头文件中声明对外接口

#endif
```

foo.c 内容

```c
// 符合：只在文件内部使用的函数的声明放在.c文件的头部，并声明为static限制其作用域
static void Bar(void);

void Foo(void)
{
    Bar();
}

static void Bar(void)
{
    ...
}
```

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。被多个源文件调用的内联函数要放在头文件中定义。

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（一个.h文件对应一个.c文件）。例如像命令 ID 定义头文件这种特别简单的头文件，不需要有对应的.c存在。

允许出现一个.h对应多个.c文件。

【例外】
main函数、单元测试函数所在的.c文件如果不需要提供对外接口，可以没有对应的.h文件。

G.INC.02 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

【描述】
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。
本条款不提倡将私有定义单独放在头文件中，具体见条款 “在头文件中声明需要对外公开的接口”。
除此之外，使用 .inc 可能还导致某些IDE工具无法识别其为头文件，造成很多功能不可用，如“跳转到变量定义处”。虽然其中一些可以通过配置强迫IDE识别 .inc 为头文件，但还是有些无法通过配置识别。

G.INC.03 禁止头文件循环依赖

【描述】
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。

G.INC.04 禁止包含用不到的头文件

【描述】
如果当前源文件没有直接引用头文件中的对外接口，则不应该包含。

包含不需要的头文件会引入不必要的依赖，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。程序员为了省事起见，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。

G.INC.05 头文件应当自包含

【描述】
简单的说，自包含就是任意一个头文件均可独立编译。如果包含某个头文件，会引入对其他头文件的依赖，则会给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：该规则需要与禁止包含用不到的头文件规则一起使用，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。

G.INC.06 头文件必须用#define保护，防止重复包含

【描述】
为防止头文件被重复包含，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 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
```

注意，保护符命名时，避免首尾是下划线(_)：

```
#ifndef _VOS_INCLUDE_TIME_H_   // 不符合
```

G.INC.07 禁止通过声明的方式引用外部函数接口、变量

【描述】
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。

【反例】
a.c 内容

```
int Foo(void);          // 不符合：通过声明的方式引用外部函数
extern int g_bar;       // 不符合：通过声明的方式引用外部全局变量

int Func(void)
{
    int i = Foo();      // 这里使用了外部接口
    ...
    return g_bar;       // 这里使用了外部变量
}
```

【正例】
a.c 内容

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

int Func(void)
{
    int i = Foo();      // 这里使用了外部接口
    ...
    return g_bar;       // 这里使用了外部变量
}
```

b.h 内容

```
int Foo(void);
extern int g_bar;
```

b.c 内容

```
int Foo(void)
{
    // 执行某些操作
}

int g_bar;
```

【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。

G.INC.08 禁止在 extern "C" 中包含头文件

【描述】
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。

【反例】
存在a.h和b.h两个头文件，按照 a.h 作者的本意，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。

```
// a.h的内容
...
#ifdef __cplusplus
void Foo(int);
#define A(value) Foo(value)
#else
void A(int);
#endif
```

```
// b.h的内容
...
#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);
}
```

【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。

G.INC.09 按照合理的顺序包含头文件

【描述】
使用固定的头文件包含顺序可增强可读性,避免隐藏依赖。
建议按稳定度包含头文件，依次顺序为：
C标准库，操作系统库，平台库，项目公共库，自己其他的依赖。

【正例】
考虑到C标准库，操作系统库等头文件比项目自研的头文件相对稳定，foo.c中包含头文件的次序如下：

```
#include <stdlib.h> // C 标准库
#include <string.h>

#include <linux/list.h> // 操作系统库
#include <linux/time.h>

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

#include "project/public/log.h" // 项目公共库

#include "bar.h" // foo.c 的依赖 bar.h

#include "foo.h" // foo.c 对应头文件放最后一个，也可以放第一个
```

【例外】
当源文件对应的头文件用于验证自包含时，可以作为第一个头文件。

2.3 数据类型

G.TYP.01 不要重复定义基础类型

【描述】
产品或项目应规划使用相同版本的类型定义，这是一种很好的实践。建议优先使用符合C99，C11标准定义的类型（如：在一个产品中统一使用stdint.h中定义的整数类型）。

避免滥用 `typedef`/`#define` 对基础类型起别名，因为一个产品或项目中如果存在多种类型定义（如同时使用u32，v32，uint32，uint表示unsigned int类型）会使代码变得更加混乱，难以维护和难以编译。使用不同的类型系统，也违反了风格一致原则。

所以除非有明确的必要性，否则不要用 `typedef`/`#define` 对基础类型进行重定义。

下面的场景中重新定义了基础类型为uintptr，其屏蔽了不同平台（CPU/OS）下C语言基本数值类型的位宽差异，是具有必要性的。

```
#include <bits/wordsize.h>

// __WORDSIZE is defined in wordsize.h
#if __WORDSIZE == 64
typedef unsigned long int uintptr;
#else
typedef unsigned int uintptr;
#endif
```

注意：
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。

- 在可预见的未来，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。**

下面例子是不同模块都使用各自的类型别名，因为历史原因，同名的类型，定义却不同。

```
// 模块 A
...
typedef unsigned long ULONG;
...
```

```
// 模块 B
...
#define ULONG UINT32
...
```

模块 B 因为是从 32 位平台移植过来，代码中把 `ULONG` 当作 32 位使用，所以移植时使用 `UINT32` 定义。
当系统运行在 `unsigned long` 为 64 位的平台上时，如果两个模块有接口交互，则这两种类型定义将引起混乱，甚至可能出现严重问题。

G.TYP.02 使用恰当的基本类型作为操作符的操作数

【描述】
操作符的操作数，应根据操作数语义使用恰当且一致的数据类型。原则包括：

- 允许布尔类型的数据用作相等类操作符、逻辑操作符、条件操作符的操作数。
- 允许无符号整数用作位操作符的操作数，而有符号整数不允许。
- 允许整数类型用作加减乘除操作符的操作数，而枚举类型不允许。
- 允许枚举类型、整数类型作为数组下标操作符的操作数。
- 不应使用浮点类型作为相等类操作符、数组下标操作符的操作数。
- 当操作符的操作数语义是数值时，应使用数值类型作为操作数，而不应使用字符类型。

条款所述操作符包括：

- 一元操作符：+、 -、 ++、 --
- 加减操作符：+、 -
- 乘除操作符：*、 /、 %
- 赋值操作符：*=、 /=、%=、 +=、 -=、 <<=、 >>=、 &=、 ^=、 |=
- 逻辑操作符：&&、 ||、!
- 关系操作符：<、 >、 <=、 >=
- 位操作符：&、 |、 ^、 ~、 <<、 >>
- 相等类操作符：==、 !=
- 条件操作符：?:
- 数组索引操作符：[]

G.TYP.03 使用合适的类型表示字符

【描述】
字符串是一个连续的字符序列，以第一个null字符结束，并且包含第一个null字符。
由于在不同系统中，char类型可以实现为signed char或unsigned char，因此不要使用char类型来表示整数。

2.4 常量

G.CNS.01 禁止使用小写字母“l”作为数值型常量后缀

【描述】
C语言标准允许使用后缀来显式指定数值型常量的类型，例如：整数常量可以使用后缀字母 l 或 L 表示常量类型 `long`，使用字母 ll 或 LL 表示 `long long`。当使用小写字母 l 指定数值型常量类型时，在视觉上容易和数字 1 混淆。为了避免这种混淆带来的误解，应使用大写字母 L 或 LL 作为数值型常量的后缀。

【反例】
如下代码示例中，宏 CONSTANT_BARRETT_REDUCTION 的值为 0x1F7011641，但是视觉上像是 0x1F701164111：

```
#define CONSTANT_BARRETT_REDUCTION 0x1F701164111
```

【正例】
如下代码示例通过使用大写字母 L 而不是小写字母 l 来消除视觉上的错觉：

```
#define CONSTANT_BARRETT_REDUCTION 0x1F7011641LL
```

G.CNS.02 不要使用难以理解的常量

【描述】
难以理解的常量，即看不懂，通过上下文也难以明确含义的数字常量、字符串常量。
难以理解的常量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，不能明确12代表什么类型；
但 `year = month * 12`；就能看懂，这里12很明显是指1年有12个月。

数字 0 有时候也是难以理解的常量，比如 `status = 0;`，0 无法明确是什么状态。

解决途径：

- 对于单点使用的难以理解的常量，按需增加注释说明。
- 对于多处使用的难以理解的常量，应该定义宏或 const 变量，并通过符号命名自注释。

禁止出现下列情况：

- 没有通过符号命名来解释数字含义，如 `#define ZERO 0`
- 符号命名限制了其取值，如 `#define XX_TIMER_INTERVAL_300MS 300`

【反例】
如下示例代码中，使用了难以理解的常量：

```
int currentType = psInParam->GetValue("servType");
// 下面使用的数字，不容易理解其具体含义
if (currentType == 1) {
    ...
} else if (currentType == 4) {
    ...
} else {
    ...
}

if (age >= 18) {
```

```
    ... // 执行某些操作
} else {
    ... // 执行另外的某些操作
}
```

【正例】
如下代码示例中，使用能表达含义的宏或常量：

```
#define SERV_TYPE_SET 1
#define SERV_TYPE_QUERY 4
#define ADULT_AGE 18

int currentType = psInParam->GetValue("servType");

if (currentType == SERV_TYPE_SET) {
    ...
} else if (currentType == SERV_TYPE_QUERY) {
    ...
} else {
    ...
}

if (age >= ADULT_AGE) {
    ... // 执行某些操作
} else {
    ... // 执行另外的某些操作
}
```

【反例】
如下示例代码中的问题在于计算长度时`strlen("V100R01C10")`中，字符串遗漏了一个'1'：

```
bCond = (strncmp(gSrcDataVer, "V100R011C10", strlen("V100R01C10")) == 0);
```

【正例】
如下代码的修改有两个好处：
1.不会出错；
2.如果版本号要修改，修改一次即可：

```
// 为字符串常量（表示版本号）重定义符号常量
#define    VERSION    "V100R011C10"
bCond = (strncmp(gSrcDataVer, VERSION, strlen(VERSION)) == 0);
```

2.5 变量

G.VAR.01 禁止读取未经初始化的变量

【描述】
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 condVal)
{
    int data;
    Bar(data); // 不符合：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：

```
#define CUSTOMIZED_SIZE 100
void Foo(int condVal)
{
    int data;
    if (condVal > 0) {
        data = CUSTOMIZED_SIZE;
    }
    Bar(data); // 不符合：其他分支中(condVal <= 0)，data值未初始化
    ...
}
```

注意：如果编译器允许，变量应按需定义，避免初始化问题。

G.VAR.02 不要在子作用域中重复使用相同变量名

【描述】
当两个作用域存在包含关系时，不要在较小的作用域内定义与较大作用域中相同的变量名，以免引起混淆。

G.VAR.03 避免大量栈分配

【描述】
程序在运行期间，函数内的局部变量存储在栈中，而栈的大小是有限的，局部变量占用的栈空间过大时，可能导致出现栈溢出错误。
建议在定义函数局部变量时(特别是递归调用、循环体中定义变量)，需要充分考虑单个函数及全调用栈的开销，并对函数中的局部变量大小进行一定限制，避免因占用过多的栈空间导致程序运行失败。

例如，linux内核的默认构建配置文件中，限制函数帧的大小不超过1024字节，如果超出限制则出现编译告警。

【反例】
如下代码示例中的`buff[MAX_BUFF]`数组占用空间过大，可能导致栈空间不够，程序发生 stack overflow 异常。

```
#define MAX_BUFF 0x1000000

char buff[MAX_BUFF] = {0};
...
```

【正例】
如下代码示例中，通过动态分配内存的方式，避免栈空间占用过大的问题：

```
#define MAX_BUFF 0x1000000
char *buf = (char *)malloc(MAX_BUFF);
if (buf == NULL) {
    ... // 错误处理
}
...
```

如上代码中须检查动态分配函数的返回值，如果分配内存大小的来源不可信，则需要注意校验其值的范围。

【反例】
递归调用太深也会造成栈分配过大，因此递归函数必须确保调用深度可控。
如下递归函数，没有控制其递归深度，调用深度随m的值增加而增加，所使用的栈大小也随之增加，可能导致栈空间不足：

```
uint64_t Sum(uint32_t m)
{
    if (m == 0) {
        return 0;
    }
    return Sum(m - 1) + m;
}
```

【正例】
重构上面的递归函数，没有使用递归算法，所使用的栈大小不会随m的变化而变化：

```
uint64_t Sum(uint32_t m)
{
    uint64_t n = (uint64_t)m;
    return (n * (n + 1) / 2);
}
```

G.VAR.04 慎用全局变量

【描述】

谨慎使用全局变量，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。

如不可避免需要使用全局变量，需要注意：

- 对全局变量的读写应集中封装。
- 避免使用全局变量作为模块接口。

G.VAR.05 指向资源句柄或描述符的变量，在资源释放后立即赋予新值

【描述】

指向资源句柄或描述符的变量包括指针、文件描述符、socket描述符以及其它指向资源的变量。
以指针为例，当指针成功申请了一段内存之后，在这段内存释放以后，如果其指针未立即设置为NULL，也未分配一个新的对象，那这个指针就是一个悬空指针。
如果再对悬空指针操作，可能会发生重复释放或访问已释放内存的问题，造成安全漏洞。
消减该漏洞的有效方法是将释放后的指针立即设置为一个确定的新值，例如设置为NULL。对于全局性的资源句柄或描述符，在资源释放后，应该马上设置新值，以避免使用其已释放的无效值；对于只在单个函数内使用的资源句柄或描述符，应确保资源释放后其无效值不被再次使用。

【反例】

如下代码示例中，根据消息类型处理消息，处理完后释放掉body指向的内存，但是释放后未将指针设置为NULL。如果还有其他函数再次处理该消息结构体时，可能出现重复释放内存或访问已释放内存的问题。

```
int Func(void)
{
    SomeStruct *msg = NULL;
    int ret = 0;

    ... // 分配msg，初始化msg->type，分配 msg->body 的内存空间

    if (msg->type == MESSAGE_A) {
        ...
        free(msg->body); // 错误：释放内存后，未置空
    }
    ...
    if (someError) {
        ...
        goto ERROR_EXIT;
    }
    // 将msg存入全局队列，后续可能使用已释放的body成员
    InsertMsgToQueue(msg);
    return ret;
ERROR_EXIT:
    ...
    free(msg->body);  // 可能再次释放了body的内存
    free(msg);
    return ret;
}
```

【正例】

如下代码示例中，立即对释放后的指针设置为NULL，避免重复释放指针。

```
int Func(void)
{
    SomeStruct *msg = NULL;
    int ret = 0;

    ... // 初始化msg->type，分配 msg->body 的内存空间

    if (msg->type == MESSAGE_A) {
        ...
```

```
        free(msg->body);
        msg->body = NULL;
    }
    ...
    if (someError) {
        ...
        goto ERROR_EXIT;
    }
    // 将msg存入全局队列，后续可能使用已释放的body成员
    InsertMsgToQueue(msg);
    return ret;
ERROR_EXIT:
    ...
    free(msg->body); // 马上离开作用域，不必赋值NULL
    free(msg);       // 马上离开作用域，不必赋值NULL
    return ret;
}
```

【反例】
如下代码示例中文件描述符关闭后未赋新值。

```
SOCKET s = INVALID_SOCKET;
int fd = -1;
...
closesocket(s);
...
close(fd);
...
```

【正例】
如下代码示例中，在资源释放后，对应的变量应该立即赋予新值。

```
SOCKET s = INVALID_SOCKET;
int fd = -1;
...
closesocket(s);
s = INVALID_SOCKET;
...
close(fd);
fd = -1;
...
```

【反例】
如下代码示例中，FreeSomeStruct函数中释放p后设置NULL的操作是无效的，导致DoSomeThing函数访问已释放内存。

```
void FreeSomeStruct(SomeStruct *p)
{
    if (p == NULL) {
        return;
    }
    if (p->content != NULL) {
        free(p->content);
        p->content = NULL;
    }
```

```
    free(p);
    p = NULL;
}

void DoSomeThing(void)
{
    SomeStruct *p = ... // 分配结构体内存并初始化
    ...
    if (condition) {
        FreeSomeStruct(p);
    }
    if (p != NULL) {
        // 可能会访问已释放内存
        errno_t ret = memcpy_s(buf, sizeof(buf), p->content, p->contentLen);
        ...
    }
}
```

【正例】
如下代码示例中，立即对释放后的指针设置为NULL，避免访问已释放内存。

```
void FreeSomeStruct(SomeStruct *p)
{
    if (p == NULL) {
        return;
    }
    if (p->content != NULL) {
        free(p->content);
        p->content = NULL;
    }
    free(p);
}

void DoSomeThing(void)
{
    SomeStruct *p = ... // 分配结构体内存并初始化
    ...
    if (condition) {
        FreeSomeStruct(p);
        p = NULL;
    }
    if (p != NULL) {
        errno_t ret = memcpy_s(buf, sizeof(buf), p->content, p->contentLen);
        ...
    }
}
```

G.VAR.06 禁止将局部变量的地址返回到其作用域以外

【描述】
局部变量的作用域是声明该变量的函数体或函数体内的语句块，声明的局部变量仅在其作用域内是可见的，局部变量生命周期始于其声明终于其作用域结束。如果对象在其生命周期之外被引用，则程序的行为是未定义的。

G.VAR.07 避免将只在一个函数中使用的变量声明为全局变量

【描述】
变量如果仅在函数范围内使用，那么它不应声明为文件范围的变量或者具有外部链接属性的全局变量。

G.VAR.08 资源不再使用时应予以关闭或释放

【描述】
这里的资源包括计算机内存、文件描述符、socket描述符。

程序员在创建或分配资源后，如果该资源不再被使用，程序员应将其正确的关闭或释放，尤其要注意所有可能的异常路径，避免遗漏。

以计算机内存为例，当动态分配内存不再使用时应予以释放，否则会发生内存泄漏，如果攻击者可以有意触发该漏洞，则内存资源可能被耗尽，造成拒绝服务。

以文件描述符为例，当打开的文件不再使用时应将指向该文件的描述符关闭，否则会发生该文件描述符泄漏，如果攻击者可以有意触发该漏洞，则可用的文件描述符可能被耗尽，造成拒绝服务。

【反例】

```
#define BLOCK_SIZE_MAX    256

char *GetBlock(int fd)
{
    ...
    char *buf = (char *)malloc(BLOCK_SIZE_MAX);
    if (buf == NULL) {
        return NULL;
    }

    if (read(fd, buf, BLOCK_SIZE_MAX) != BLOCK_SIZE_MAX) {
        return NULL;  // 错误：在异常路径中返回前未释放buf指向的内存资源，存在内存泄漏。
    }
    return buf;
}
```

【正例】

```
#define    BLOCK_SIZE_MAX    256

char *GetBlock(int fd)
{
    ...
    char *buf = (char *)malloc(BLOCK_SIZE_MAX);
    if (buf == NULL) {
        return NULL;
    }

    if (read(fd, buf, BLOCK_SIZE_MAX) != BLOCK_SIZE_MAX) {
        free(buf);  // 符合：在异常路径中返回前释放pBuf指向的内存资源，并立即将指针置空。
        buf = NULL;
    }
    return buf;
}
```

2.6 表达式

G.EXP.01 执行算术运算或比较操作的两个操作数要求具有相同的类型

【描述】
在执行算术运算或者比较操作时，C语言允许不同类型操作数之间进行转换，包括显式转换和隐式转换。然而，不管是显式转换还是隐式转换，均可能会导致以下几个问题：数值丢失、符号丢失、精度丢失、布局丢失。这就要求程序员必须明确识别类型转换之间的任何潜在风险，而显示转换肯定比隐式转换更容易识别风险，相同的类型之间的算术运算或比较操作肯定比显示转换更加容易被识别风险。

G.EXP.02 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

【描述】
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)` 更是难以理解。

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) ...
```

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。

【例外】
用来描述数值区间时，可以写成 `if (MIN < v && v < MAX)`

G.EXP.03 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

【描述】
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语言标准中未明确定义。不同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。

错误写法：

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

函数参数：

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

G.EXP.04 用括号明确表达式的操作顺序，避免过分依赖默认优先级

【描述】
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，并降低了可读性，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，比如表达中同时包含位操作符和其他类型操作符。

【正例】

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

G.EXP.05 不要向sizeof传递有副作用的操作数

【描述】
不要向sizeof传递有副作用的操作数，因为操作数的副作用不一定会发生。
以sizeof(expr)为例，如果expr表达式的计算结果不影响sizeof(expr)语句的结果，则是否对expr表达式求值的行为未指定。

G.EXP.06 避免依赖结构体中位域的存储布局

【描述】
结构体中位域的存储布局是与实现相关的，虽然在每种实现中的布局是确定的，但是在不同的实现下通常不具备可移植性。因此，不能依赖于位域的存储布局，否则代码将不可移植，并且可能引发难以检测的错误。

2.7 控制语句

2.7.1 判断语句

G.CTL.01 控制表达式的结果必须是布尔值

【描述】
使用强类型可以减少C语言的编程风险，控制表达式的结果必须是布尔值以及如下表达式的运算结果：

- 关系表达式　　<　<=　>=　>
- 相等类表达式　==　!=
- 逻辑表达式　　&&　||　!

控制表达式应用的语句包括：

- if语句
- while语句。
- do语句
- for语句

```
// if语句
if (controlling expression) {
    ...
}
```

```
// while语句
while (controlling expression) {
    ...
}

// do语句
do {
    ...
} while (controlling expression);

// for语句
for ( ...; controlling expression; ... ) {
    ...
}
```

【反例】
如下代码示例不符合条款要求。

```
int value = GetValue();
if (value) { // 不符合：不是布尔类型变量
    ...
}
```

【正例】

```
int value = GetValue();
if (value != 0) { // 符合：使用的是逻辑表达式
    ...
}
```

```
while (true) { // 符合：特殊场景下可以使用布尔类型常量
    ...
}
```

```
char *p = GetPointer();
if (p != NULL) { // 符合
    ...
}
```

```
char *p = GetPointer();
if (p == NULL) { // 符合
    ...
}
```

【例外】
当判断条件中的表达式是指针时，可以使用`if (p)`表达“如果p是有效指针”的含义，因此允许写成`if (p)`的形式。

```
char *p = GetPointer();
if (p) { // 符合
    ...
}
```

```
char *p = GetPointer();
if (!p) { // 允许使用
    ...
}
```

G.CTL.02 &&和||操作符的右侧操作数不应包含副作用

【描述】
逻辑与（&&）、逻辑或（||）表达式中的右操作数是否被求值，取决于左操作数的求值结果，当左操作数的求值结果可以得出整个逻辑表达式的结果时，不会再计算右操作数的结果。如果右操作数包含副作用，则不能确定是否确实发生了副作用，因此，本规范中建议逻辑与（&&）、逻辑或（||）操作符的右操作数中不要含有副作用。

【反例】
如下代码示例中，当`flag > 0`或`value > 0`时进入分支处理，但是 value 自减的前提是`flag <= 0`。尽管代码行为是正确的，也有可能是精心设计的，但是不易阅读理解，并且给维护带来不便，容易引入问题：

```
if (flag > 0 || value-- > 0) {
  ...
}
```

【正例】
如下代码示例中，明确逻辑行为（可以将公共代码提取成函数在不同分支调用）：

```
if (flag > 0) {
    ...
} else {
    value--;
    if (value >= 0) {
        ...
    }
}
```

【反例】
如下代码示例中，Call()函数中的逻辑与（&&）操作符的右操作数调用了一个导致副作用的函数：

```
#define FILENAME_LEN 128
static int RemoveFile(unsigned int fileId)
{
    char filename[FILENAME_LEN];
    int ret = sprintf_s(filename, FILENAME_LEN, "/some_dir/%u.txt", fileId);
    if (ret < 0) {
        return -1;
    }
    return unlink(filename); // 该语句具有副作用
}

void Call(void)
{
    ...
    if (isRight && RemoveFile(fileId) != 0) {
        ...
```

```
    }
}
```

【正例】
如下的一个解决方案是拆分if语句中的表达式：

```
#define FILENAME_LEN 128
static int RemoveFile(unsigned int fileId)
{
    char filename[FILENAME_LEN];
    int ret = sprintf_s(filename, FILENAME_LEN, "/some_dir/%u.txt", fileId);
    if (ret < 0) {
        return -1;
    }
    return unlink(filename); // 该语句具有副作用
}

void Call(void)
{
    ...
    if (isRight) {
        if (RemoveFile(fileId) != 0) {
            ...
        }
    }
}
```

2.7.2 循环语句

G.CTL.03 循环必须安全退出

【描述】
在应用程序中，一个重复提供服务的逻辑循环应当设计退出机制，并且将资源正确释放后安全退出。退出条件的设计，除了让程序逻辑更加完整，也能通过实现优雅退出的代码，显式释放服务循环中分配的资源，避免资源泄漏。

【反例】
以下代码，在一个大循环内，ReceiveMsg函数内申请资源，接收外部数据。ParseMsg函数内处理数据，但没有退出条件，会导致循环前申请的资源无法释放，该程序没有安全退出。

```
...

void DoService(void)
{
    ...
    size_t size = 0;
    unsigned char *pMsg = NULL;

    CreateServiceResource(); // 分配服务资源
    while (true) {
        pMsg = ReceiveMsg(&size);
        if (pMsg != NULL) {
            ParseMsg(pMsg, size);
            FreeMsg(pMsg);
        }
        size = 0;
```

```
    }
}
```

【正例】
重新设计函数，通过提供服务退出条件，并在资源释放函数ReleaseServiceResource内释放服务循环前申请的资源。

```
bool ParseMsg(unsigned char *msg, size_t msgLen)
{
    ...
    if (msg->type == EXIT_MESSAGE_TYPE) {
        return false;
    } else {
        return true;
    }
}

void DoService(void)
{
    ...
    size_t size = 0;
    unsigned char *pMsg = NULL;
    bool doRunServiceFlag = true;    // 服务退出条件

    CreateServiceResource(); // 分配服务资源
    while (doRunServiceFlag) {
        pMsg = ReceiveMsg(&size);
        if (pMsg != NULL) {
            doRunServiceFlag = ParseMsg(pMsg, size);
            FreeMsg(pMsg);
        }
        size = 0;
    }
    ReleaseServiceResource(); // 释放服务资源
}
```

【例外】

1、操作系统软件的IDLE线程，可能需要无限循环

2、操作系统在不可恢复的错误中，为避免更多错误发生，进入指令无限循环。例如：

```
void FaultReboot(void)
{
    ...
    reboot(rebootCmd);
    while (1) {
        CoreWait();
    }
}
```

3、嵌入式设备的操作系统或主流程，可能使用无限循环。例如：

```
void OsTask(void)
{
    ...
```

```
    while (1) {
        msgHdl = Receive(OS_WAIT_FOREVER, &msgId, &senderPid);
        if (msgHdl == 0) {
            continue;
        }
        switch (msgId) {
            ...
        }
        (void)Free(msgHdl);
        msgHdl = NULL;
    }
}
```

G.CTL.04 禁止使用浮点数作为循环计数器

【描述】
二进制浮点数算数标准ISO/IEEE Std 754-1985中规定了32位单精度和64位双精度浮点类型的表示方法。因为存储二进制浮点的bit位是有限的，所以二进制浮点数的表示范围也是有限的，并且无法精确地表示所有实数。因此，浮点数计算结果也不是精确值，不能将浮点数用作循环计数器。

2.7.3 goto语句

G.CTL.05 慎用 goto 语句

【描述】
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内 goto 语句之后的标签。

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，例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（例如关闭文件句柄，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。

实际也可以将失败处理的代码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。

【代码示例】

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

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

G.CTL.06 goto语句只能向下跳转

【描述】
goto语句会破坏程序的结构性，特别是使用往回跳的goto语句，会增加代码的复杂性，使程序结构难以理解，所以除非确实需要，最好不使用goto语句。如果使用时，必须加以限制，只允许在本函数内跳转且只允许向goto语句之后的标签跳转。

2.7.4 switch语句

G.CTL.07 switch语句要有default分支

【描述】
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。使用时统一将default分支放到语句块的最后位置。

【例外】
如果switch条件变量是枚举类型，并且 case分支覆盖了所有取值，则可以不要求有default分支。

G.CTL.08 switch语句中至少有两个条件分支

【描述】
单个路径的选择语句更适合用`if`语句进行判断；且如果条件分支是布尔值，那么也不合适使用`switch`语句，使用`if`语句更合适。

【正例】

```
void Foo(void)
{
    ...

    switch (lightColor) {
        case RED:
            lastSeconds = 30;    //  红灯保持时间
            break;
        case GREEN:
            lastSeconds = 45;    // 绿灯保持时间
            break;
        default: // 符合: 存在两个分支
```

```
            lastSeconds = 3;     // 除了红灯和绿灯，其他（包括黄灯）保持3秒
            break;
    }
}
```

2.8 声明与初始化

G.DCL.01 不要声明或定义保留的标识符

【描述】
如果声明或者定义了一个保留的标识符，那么程序的行为是未定义的。

【反例】

```
#undef __LINE__                 // 不符合：两个下划线开始的标识符是保留标识符
#define _MODULE_INCLUDE_        // 不符合：下划线和一个大写字母开始的标识符
int errno;                      // 不符合：errno是标准库中的保留标识符
void *malloc(size_t nbytes);    // 不符合：malloc是标准库中的保留标识符
#define SIZE_MAX 80             // 不符合：SIZE_MAX是标准库中的保留宏定义
```

2.9 整数

C语言中整数类型繁多，包括有符号、无符号以及不同大小的整数类型，在不同实现中相同类型的整数的表示形式可能不同，并且不同整数类型间转换规则非常复杂，因此在使用整数前需要了解整数提升规则，了解不同整数类型间转换规则（隐式转换规则、显示转换规则），了解整数常量/表达式中容易出现的问题，以及算数运算导致的整数溢出（overflow）/整数回绕（wrap）所产生的问题，便于设计安全的算术运算。

G.INT.01 确保有符号整数运算不溢出

【描述】
有符号整数溢出在C语言标准中是一种未定义行为。使用溢出后的数值可能导致程序缓冲区读写越界等风险。出于安全考虑，对外部数据（参见对所有外部数据进行合法性检查）中的有符号整数值在如下场景中使用时，需要确保运算不会导致溢出：

- 指针偏移值（指针算术运算的整数操作数）
- 数组索引值
- 内存拷贝的长度
- 内存分配函数的参数
- 循环判断条件

【反例】
如下代码示例中，参与减法运算的整数是外部数据，在使用前未做校验，可能出现整数溢出，进而造成后续的内存复制操作出现缓冲区溢出。

```
unsigned char *content = ...  // 指向报文头的指针
size_t contentSize = ...      // 缓冲区的总长度
int totalLen = ...            // 报文总长度
int skipLen = ...             // 从消息中解析出来的需要忽略的数据长度

// 用 totalLen - skipLen 计算剩余数据长度，可能出现整数溢出
errno_t ret = memmove_s(content,
                        contentSize,
                        content + skipLen,
                        totalLen - skipLen);
...
```

【正例】
如下代码示例中，重构为使用`size_t`类型的变量表示数据长度，并校验外部数据长度是否在合法范围内。

```
unsigned char *content = ... //指向报文头的指针
size_t contentSize = ...     // 缓冲区的总长度
size_t totalLen = ...        // 报文总长度
size_t skipLen = ...         // 从消息中解析出来的需要忽略的数据长度

if (skipLen >= totalLen || totalLen > contentSize) {
    ... // 错误处理
}
errno_t ret = memmove_s(content,
                        contentSize,
                        content + skipLen,
                        totalLen - skipLen);
...
```

【反例】
如下代码示例中，内核代码对来自用户态的数值范围做了校验，但是由于opt是`int`类型，而校验条件中错误的使用了`ULONG_MAX`进行限制，导致整数溢出。

```
int opt = ... // 来自用户态
if ((opt < 0) ||
    (opt > (ULONG_MAX / (60 * HZ)))) { // 错误的使用了ULONG_MAX做上限校验
    ... // 错误处理
}
... = opt * 60 * HZ; // 可能出现整数溢出
...
```

【正例】
一种改进方案是将opt的类型修改为`unsigned long`类型，这种方案适用于修改了变量类型更符合业务逻辑的场景。

```
unsigned long opt = ... // 将类型重构为 unsigned long 类型。
if (opt > (ULONG_MAX / (60 * HZ))) {
    ... // 错误处理
}
... = opt * 60 * HZ;
...
```

另一种改进方案是将数值上限修改为INT_MAX。

```
int opt = ... // 来自用户态
if ((opt < 0) ||
    (opt > (INT_MAX / (60 * HZ)))) { // 修改使用 INT_MAX作为上限值
    ... // 错误处理
}
... = opt * 60 * HZ;
```

G.INT.02 确保无符号整数运算不回绕

【描述】

无符号整数的算术运算结果可能会发生整数回绕。使用回绕后的数值其可能导致程序缓冲区读写越界等风险。出于安全考虑，对外部数据（参见对所有外部数据进行合法性检查）中的无符号整数值在如下场景中使用时，需要确保运算不会导致回绕：

- 指针偏移值（指针算术运算的整数操作数）
- 数组索引值
- 内存拷贝的长度
- 内存分配函数的参数
- 循环判断条件

【反例】

如下代码示例中，校验下一个子报文的长度加上已处理报文的长度是否超过了整体报文的最大长度，在校验条件中的加法运算可能会出现整数回绕，造成绕过该校验的问题。

```
size_t totalLen = ... // 报文的总长度
size_t readLen = 0 // 记录已经处理报文的长度
...
size_t pktLen = ParsePktLen(); // 从网络报文中解析出来的下一个子报文的长度
if (readLen + pktLen > totalLen) { // 可能出现整数回绕
  ... // 错误处理
}
...
readLen += pktLen;
...
```

【正例】

由于readLen变量记录的是已经处理报文的长度，必然会小于totalLen，因此将代码中的加法运算修改为减法运算，导致条件绕过。

```
size_t totalLen = ... // 报文的总长度
size_t readLen = 0; // 记录已经处理报文的长度
...
size_t pktLen = ParsePktLen(); // 来自网络报文
if (pktLen > totalLen - readLen) {
  ... // 错误处理
}
...
readLen += pktLen;
...
```

【反例】

如下代码示例中，校验len合法范围的运算可能会出现整数回绕，导致条件绕过。

```
size_t len = ... // 来自用户态输入
if (SCTP_SIZE_MAX - len < sizeof(SctpAuthBytes)) { // 减法操作可能出现整数回绕
    ... // 错误处理
}
... = kmalloc(sizeof(SctpAuthBytes) + len, gfp); // 可能出现整数回绕
...
```

【正例】
如下代码示例中，调整减法运算的位置（需要确保编译期间减法表达式的值不回绕），避免整数回绕问题。

```
size_t len = ... // 来自用户态输入
if (len > SCTP_SIZE_MAX - sizeof(SctpAuthBytes)) { // 确保编译期间减法表达式的值不翻转
    ... // 错误处理
}
... = kmalloc(sizeof(SctpAuthBytes) + len, gfp);
...
```

G.INT.03 确保除法和余数运算不会导致除零错误(被零除)

【描述】
如果整数的除法运算或取余运算的除数为0会导致程序产生未定义的行为。对涉及到除法或者取余运算，必须确保除数不为0。

【反例】

```
size_t a = ReadSize();
size_t b = 1000 / a;    // 不符合：a可能是0
size_t c = 1000 % a;    // 不符合：a可能是0
...
```

【正例】
如下代码示例中，添加a是否为0的校验，防止除零错误。

```
size_t a = ReadSize();
if (a == 0) {
    ... // 错误处理
}
size_t b = 1000 / a;    // 符合：确保a不为0
size_t c = 1000 % a;    // 符合：确保a不为0
...
```

G.INT.04 整型表达式比较或赋值为一种更大类型之前必须用这种更大类型对它进行求值

【描述】
由于整数在运算过程中可能出现有符号整数溢出、无符号整数回绕等问题，当运算结果赋值给比它更大的类型，或者与比它更大的类型进行比较时，可能会导致实际结果与预期结果不符。
如果将涉及某个操作的整数表达式与较大的整数大小进行比较或分配给较大的整数，则该整数表达式应该通过显式转换其中一个操作数来以较大类型的大小进行计算。
类似的，当组合表达式的运算结果赋值给比它更大类型，或者与比它更大类型进行运算时，应显式转换其中一个操作数为较大的类型。

在int为32位，long long为64位并以二进制补码表示整数的系统中，请观察以下二个代码及其输出，以便了解本规则所解决的问题：

```
int main(int argc, char *argv[])
{
    unsigned int a = 0x10000000;
    unsigned long long b = a * 0xab;
    printf("b = %llX\n", b);
    return 0;
}
```

输出：

```
b = B0000000
```

```
int main(int argc, char *argv[])
{
    unsigned int a = 0x10000000;
    unsigned long long b = (unsigned long long )a * 0xab;
    printf("b = %llX\n", b);
    return 0;
}
```

输出：

```
b = AB0000000
```

【反例】（组合表达式）

以二进制补码表示整数的系统中，如下代码示例，
表达式 `a + b + c` 等同于表达式 `(a + b) + c`，计算组合表达式 `(a + b)` 时先发生整数回绕，其结果为0，再与c相加后得到 x 的值为2。
表达式 `(uint64_t)(a + b) + c` ，计算组合表达式 `(a + b)` 时先发生整数回绕，其结果为0，然后再转换 `uint64_t` 类型并与 c 相加后得到 y 的值为2。
表达式 `a + c + b` 等同于表达式 `(a + c) + b`，与前面表达式不同点在于，计算 `a + c` 时发生了隐式类型转换，将 a 隐式提升到了 `uint64_t` 类型后在进行计算，因此最后 z 的值是正确的，但是该代码依赖表达式计算顺序，不利于维护和阅读，应禁止使用该技巧。类似的，t 和 m 的值虽然正确，但是依赖表达式优先级，应禁止使用该技巧。

```
uint32_t a = 0xffffffffU;
uint32_t b = 1;
uint64_t c = 2;
uint64_t x = a + b + c;   // 结果非预期，x的值为2
uint64_t y = (uint64_t)(a + b) + c; // 结果非预期，y的值为2
uint64_t z = a + c + b;   // 结果正确，但是依赖表达式顺序，禁止使用该技巧
uint64_t t = a + (b + c); // 结果正确，但是依赖表达式优先级，禁止使用该技巧
uint64_t m = a + b * c;   // 结果正确，但是依赖表达式优先级，禁止使用该技巧
```

【正例】（组合表达式）

如下的一种解决方案，是显式转换变量 a 和 b 的类型为 `uint64_t` 类型：

```
uint32_t a = 0xffffffffU;
uint32_t b = 1;
uint64_t c = 2;
uint64_t x = (uint64_t)a + (uint64_t)b + c;
uint64_t y = ((uint64_t)a + (uint64_t)b) + c;
uint64_t z = (uint64_t)a + c + (uint64_t)b;
uint64_t t = (uint64_t)a + ((uint64_t)b + c);
uint64_t m = (uint64_t)a + (uint64_t)b * c;
```

最佳做法是修改变量 a 和 b 的类型为 `uint64_t` 类型，保持表达式中的操作数类型一致（如果表达式的数值不可信，应先校验其范围防止无符号整数回绕）：

```
uint64_t a = 0xffffffffU;
uint64_t b = 1;
uint64_t c = 2;
uint64_t x = a + b + c;
uint64_t y = a + c + b;
uint64_t z = a + (b + c);
uint64_t t = a + b * c;
```

G.INT.05 只能对无符号整数进行位运算

【描述】
因为对有符号整数进行位运算的结果是由实现定义的，所以只能对无符号整数进行位运算。此外，对精度低于int类型的无符号整数进行位运算时，编译器会进行整数提升，再对提升后的整数进行位运算，因此要特别注意对于这类无符号整数的位运算，避免出现非预期的结果。
本条款涉及的位操作符包括：

- ~（求反）
- &（与）
- |（或）
- ^（异或）
- >>（右移位）
- <<（左移位）
- &=
- ^=
- |=
- >>=
- <<=

【反例】

```
int data = ReadByte();
int value = data >> 28;           //  对有符号整数进行位运算，程序的行为是未定义的

... // 检查 data 的合法范围，代码略

int mask = 1 << data;            //  对有符号整数进行位运算，程序的行为是未定义的
```

【正例】

```
unsigned int data = (unsigned int)ReadByte();
unsigned int value = data >> 28; // 只能对无符号整数进行位运算

... // 检查 data 的合法范围，代码略

unsigned mask  = 1 << data;
```

G.INT.06 校验外部数据中整数值的合法性

【描述】
外部数据中的整数值是不可信的，必须校验其合法性，防止发生各种潜在风险。

【反例】
如下示例中，由于循环条件受外部输入的报文内容控制，可能进入无限循环从而导致拒绝服务：

```
unsigned char *FindAttr(unsigned char type, const unsigned char *msg, size_t
inputMsgLen)
{
    const unsigned char *content = msg;
    ...
    contentLength = content[RD_LEA_PKT_LENGTH]);
    +...
    while (contentLength < RD_LEA_PKT_LENGTH + 1) {
        mAttrType = content[0];
        mAttrLength = content[RD_LEA_PKT_LENGTH];
        ...
        contentLength -= mAttrLength;
        content += mAttrLength;
    }
    ...
}
```

【正例】
如下示例中，通过检查消息剩余长度，避免后续运算中无符号整数回绕。

```
unsigned char *FindAttr(unsigned char type, const unsigned char *msg, size_t
inputMsgLen)
{
    const unsigned char *content = msg;
    ...
    contentLength = content[RD_LEA_PKT_LENGTH]);
    ...
    while (contentLength < RD_LEA_PKT_LENGTH + 1) {
        mAttrType = content[0];
        mAttrLength = content[RD_LEA_PKT_LENGTH];
        ...
        if (contentLength < mAttrLength) {  // 检查消息剩余长度
            ... // 错误处理
            break;
        }
        contentLength -= mAttrLength;
        content += mAttrLength;
    }
    ...
}
```

G.INT.07 移位操作符的右操作数必须是非负数且小于左操作数类型的位宽

【描述】
移位操作符的右操作数必须是非负数，且右操作数值必须小于左操作数类型的位宽，否则程序的行为是未定义的。例如：当移位操作的左操作数为32位无符号整数类型时，确保右操作数的取值范围是`[0,31]`。

【反例】

```
uint32_t input = ...
uint32_t value = input << 32;    // 不符合：右操作数不满足位宽要求
...
```

【正例】

```
uint32_t input = ...
uint64_t value = (uint64_t)input << 32; // 符合
...
uint32_t count = input >> 24;           // 符合
```

G.INT.08 表示对象大小的整数值或变量应当使用size_t类型

【描述】

表示对象大小的整数值或变量使用size_t类型，可以满足对象大小的精度要求，预防整数使用错误，如整数溢出、数据截断。

G.INT.09 确保枚举常量映射到唯一值

【描述】
C语言的枚举类型是一组枚举常量（整数常量）的集合。C语言没有限制枚举常量值唯一性，但本条款要求在一个枚举类型中枚举常量值是唯一的，因为在一个枚举类型中，如果存在相同的枚举常量值可能会导致程序异常而不易被发现。

【反例】
如下代码示例中，在枚举类型`SomeModuleErrCodes`中定义错误码的时候，为两个枚举常量分配了显式值，造成`ERR_SYS_MODIFY`和`ERR_EMG_INVALID`被隐式声明为相同的值（0x3a020010），对于程序员来说，可能并不明显。
这种定义可能导致的错误是尝试将枚举常量用于`switch`语句的标签。由于`switch`语句中的所有标签都必须是唯一的，因此如下代码违反了此语义约束。

```
typedef enum {
    ERR_SYS_ARG = 0x3a020000,
    ERR_SYS_SET,
    ERR_SYS_OCCUPIED,
    ERR_SYS_REQFREE,
    ERR_SYS_NO_REC,
    ERR_SYS_ACTIVED,
    ERR_SYS_NO_FILE,
    ERR_SYS_OPEN,
    ERR_SYS_ASSIGN,
    ERR_SYS_ALLOC,
    ERR_SYS_GET_LIST,
```

```
    ERR_SYS_IGNORE,
    ERR_SYS_REG,
    ERR_SYS_CALC,
    ERR_SYS_SNED,
    ERR_SYS_RECV,
    ERR_SYS_MODIFY,     // 不符合：值与ERR_EMG_INVALID相同
    ERR_SYS_CHECK,      // 不符合：值与ERR_EMG_TYPE相同

    ERR_EMG_INVALID = 0x3a020010,
    ERR_EMG_TYPE,
    ...
} SomeModuleErrCodes;
```

【正例】
为了防止出现错误代码示例中的问题，枚举类型声明可以采用以下形式之一：

- 不提供显式的整数赋值，如下示例所示

```
typedef enum {
    ERR_SYS_ARG,
    ERR_SYS_SET,
    ERR_SYS_OCCUPIED,
    ERR_SYS_REQFREE,
    ERR_SYS_NO_REC,
    ERR_SYS_ACTIVED,
    ERR_SYS_NO_FILE,
    ERR_SYS_OPEN,
    ERR_SYS_ASSIGN,
    ERR_SYS_ALLOC,
    ERR_SYS_GET_LIST,
    ERR_SYS_IGNORE,
    ERR_SYS_REG,
    ERR_SYS_CALC,
    ERR_SYS_SNED,
    ERR_SYS_RECV,
    ERR_SYS_MODIFY,
    ERR_SYS_CHECK,

    ERR_EMG_INVALID,
    ERR_EMG_TYPE,
    ...
} SomeModuleErrCodes;
```

- 只对第一个成员赋值，如下示例所示

```
typedef enum {
    ERR_SYS_ARG      = 0x3a020000,
    ERR_SYS_SET,
    ERR_SYS_OCCUPIED,
    ERR_SYS_REQFREE,
    ERR_SYS_NO_REC,
    ERR_SYS_ACTIVED,
    ERR_SYS_NO_FILE,
    ERR_SYS_OPEN,
    ERR_SYS_ASSIGN,
    ERR_SYS_ALLOC,
```

```
    ERR_SYS_GET_LIST,
    ERR_SYS_IGNORE,
    ERR_SYS_REG,
    ERR_SYS_CALC,
    ERR_SYS_SNED,
    ERR_SYS_RECV,
    ERR_SYS_MODIFY,
    ERR_SYS_CHECK,

    ERR_EMG_INVALID,
    ERR_EMG_TYPE,
    ...
} SomeModuleErrCodes;
```

在上面的两个选项中，除非第一个枚举数必须具有非零值，否则第一种做法是最简单的方法，因此也是首选方法。

【例外】
如果枚举类型的多个成员确实需要分配相同的值，需要提供显式的整数赋值，并写一条注释，解释为什么这样做，避免将来的维护人员误认为此代码有问题。

G.INT.10 确保整数转换不会造成数据截断或符号错误

【描述】
不同整数类型间相互转换时，应确保转换后的结果不会造成数据截断或符号错误。将整数转换为宽度较小的类型会导致高位被截断，有符号/无符号整数间转换可能导致符号错误。因此，无符号数转无符号数或无符号数转有符号数时，确保数值在目标类型的上限范围内；有符号数转有符号数或有符号数转无符号数时，确保数值在目标类型的上限和下限范围内。
同时，程序员也要知道：将整数值转换为具有同符号的更宽类型，以及相同类型的有符号非负数转换为无符号类型时，可以安全地转换。

【反例】
如下代码示例中，将 long 类型转换为 `size_t` 类型，属于从有符号类型的值转换为无符号类型值，可能会发生类型范围错误，如果 len 的值小于0，会发生符号错误：

```
void Func(unsigned char *msg, long len)
{
    unsigned char *buffer = NULL;
    if (len == 0) {
        ... // 处理错误
    }
    buffer = (unsigned char *)malloc((size_t)len);
    ...
}
```

【正例】
如下代码示例中，优选重构函数 len 参数的类型为 `size_t` 类型。

```
void Func(unsigned char *msg, size_t len)
{
    unsigned char *buffer = NULL;
    ...
    buffer = (unsigned char *)malloc(len); // len已进行合法性检查
    ...
}
```

2.10 指针和数组

G.ARR.01 外部数据作为数组索引时必须确保在数组大小范围内

【描述】
外部数据作为数组索引对内存进行访问时，必须对数据的大小进行严格的校验，确保数组索引在有效范围内，否则会导致严重的错误。
当一个指针指向数组元素时，可以指向数组最后一个元素的下一个元素的位置，但是不能读写该位置的内存。

【反例】
如下代码示例中，SetDevId()函数存在差一错误，当 index 等于 `DEV_NUM` 时，恰好越界写一个元素；同样GetDev()函数也存在差一错误，当解引用这个函数返回的指针时，程序的行为是未定义的。

```
...
#define DEV_NUM 10
#define MAX_NAME_LEN 128
typedef struct {
    int id;
    char name[MAX_NAME_LEN];
} Dev;

static Dev devs[DEV_NUM];

int SetDevId(size_t index, int id)
{
    if (index > DEV_NUM) {    // 差一错误。
        ... // 错误处理
    }

    devs[index].id = id;
    return 0;
}

static Dev *GetDev(size_t index)
{
    if (index > DEV_NUM) {    // 差一错误。
        ... // 错误处理
    }

    return &devs[index];
}
```

【正例】
如下代码示例中，修改校验索引的条件，避免差一错误。

```
#define DEV_NUM 10
#define MAX_NAME_LEN 128
typedef struct {
    int id;
    char name[MAX_NAME_LEN];
} Dev;

static Dev devs[DEV_NUM];
```

```
int SetDevId(size_t index, int id)
{
    if (index >= DEV_NUM) {
        ... // 错误处理
    }
    devs[index].id = id;
    return 0;
}

static Dev *GetDev(size_t index)
{
    if (index >= DEV_NUM) {
        ... // 错误处理
    }
    return &devs[index];
}
```

G.ARR.02 禁止通过对数组类型的函数参数变量进行sizeof来获取数组大小

【描述】

函数参数列表中声明为数组的参数会被调整为相应类型的指针。当将sizeof应用于声明为数组类型的形参时，sizeof操作符将得出调整后（指针）类型的大小。

例如：`void ArrayInit(int inArray[ARRAY_MAX_LEN])`函数参数列表中的inArray虽然被声明为数组，但是实际上会被调整为指向int类型的指针，即调整为`void ArrayInit(int *inArray)`。在这个函数内使用`sizeof(inArray)`等同于`sizeof(int *)`，得到的结果通常与预期不相符。

【反例】

如下代码示例中，函数内使用`sizeof(inArray)`不等于`ARRAY_MAX_LEN * sizeof(int)`。

```
#define ARRAY_MAX_LEN 256

void ArrayInit(int inArray[ARRAY_MAX_LEN])
{
    // 不符合：sizeof(inArray)结果是指针大小，不是数组大小，和预期不符。
    size_t arrayLen = sizeof(inArray) / sizeof(inArray[0]);
    ...
}
```

【正例】

如下代码示例中，使用入参len表示指定数组的长度：

```
// 函数说明：入参len是入参数组的长度
void ArrayInit(int inArray[], size_t len)
{
    ...
}
```

G.ARR.03 禁止通过对指针变量进行sizeof操作来获取数组大小

【描述】
将指针当做数组进行sizeof操作时，会导致实际的执行结果与预期不符。例如：变量定义`char *p = array`，其中array的定义为`char array[LEN]`，表达式`sizeof(p)`得到的结果与`sizeof(char *)`相同，并非array的长度。

【反例】
如下代码示例中，buffer和path分别是指针和数组，程序员意图是对这2个内存进行清0操作，但由于疏忽，将内存大小误写成了`sizeof(buffer)`，与预期不符。

```
char path[MAX_PATH];
char *buffer = (char *)malloc(SIZE);
...

...
memset(path, 0, sizeof(path));

// sizeof与预期不符，其结果为指针本身的大小而不是缓冲区大小
memset(buffer, 0, sizeof(buffer));
```

【正例】
如下代码示例中，将`sizeof(buffer)`修改为申请的缓冲区大小：

```
char path[MAX_PATH];
char *buffer = (char *)malloc(SIZE);
...

...
memset(path, 0, sizeof(path));
memset(buffer, 0, SIZE); // 使用申请的缓冲区大小
```

G.ARR.04 避免整数与指针间的互相转化

【描述】
指针的大小随着平台的不同而不同，强行进行整数与指针间的互相转化，降低了程序的兼容性，在转换过程中可能引起指针高位信息的丢失。在linux下，将指针转换为long类型之后再转换为原类型的做法通常不会丢失信息，但C语言标准并未对此予以保证。任何指向void的有效指针可以转换成intptr_t或uintptr_t类型后再转换成void指针，转换结果应与原始指针相等。
因此，当代码中出现指针和整数互转的情况，首先考虑通过修改代码避免转换，如果必须转换，建议先将指针转换为`void *`后再转换为`uintptr_t`或`intptr_t`类型存放转换后的指针值。

【反例】
如下代码示例中，在指针为64位，int为32位的64位Linux系统中，转换后的数值可能不在unsigned int类型的值域范围内，导致错误。

```
char *p = ...;
char *p2 = ...;

// 直接将指针转换为 int 可能会超出int表示范围导致错误
unsigned int number = (unsigned int)p;

// 将uintptr_t转换为int 可能会发生数据截断导致错误
unsigned int number2 = (unsigned int)(uintptr_t)p2;
```

【正例】
如下代码示例中，使用uintptr_t类型接收转换后的指针，先转换为`void *`的原因是因为C语言标准中只规定了`void *`转换`intptr_t`或`uintptr_t`的行为是确定的。

```
char *p = ...;
char *p2 = ...;
uintptr_t number = (uintptr_t)(void *)p;
uintptr_t number2 = (uintptr_t)(void *)p2;
```

G.ARR.05 不同类型的对象指针之间不应进行强制转换

【描述】
不同的对象类型可能有不同的对齐要求，如果在不同类型的对象指针之间做强制转换，或转化为void指针后再转换为不同类型的对象指针，对象的对齐方式可能被改变，从而导致程序产生未定义行为。
另外，从void指针转换为特定类型的指针是允许的，但需要满足如下要求：
1、确保转换后的指针正确对齐；
2、void指针指向的数据长度必须满足目标类型大小的要求。

【反例】

如下代码示例中，`char`类型指针`&c`被转换为更严格对齐的`int`类型指针`intPtr`再被强制转换为`char`类型指针`charPtr`。在某些实现上，`charPtr`将不匹配`&c`。因此，如果将一个对象类型的指针转换为另一个对象类型的指针，则第二个对象类型的对齐要求不能比第一个更加严格：

```
char c = 'x';               // 变量c的地址可能不在int对齐(通常为4字节对齐)的内存边界上
int *intPtr = (int *)&c;    // 不兼容的转换
char *charPtr = (char *)intPtr;

ASSERT(charPtr == &c);    // 在一些系统下会因地址不对齐失败
```

【正例】
如下代码示例中，`char`类型值被保存在一个类型为`int`的对象中，这样指针的值会被正确对齐：

```
char c = 'x';
int i = c;
int *intPtr = &i;

ASSERT(intPtr == &i);
```

【反例】
C标准允许将任何对象指针和`void *`相互转换。因此，一种指针类型可以转换为`void *`后再转换为另一种类型，即使类型不兼容也不会出现编译告警。
如下代码示例中，向 Func() 函数传入了`char`类型指针，但是函数将其转换为`int`类型指针返回，intPtr 可能比 charPtr 对齐更严格：

```
int *Func(void *ptr)
{
    ...
    return ptr;
}

void Caller(char *charPtr)
{
    int *intPtr = Func(charPtr);  // 程序可能产生未定义行为
    ...
}
```

【正例】
如下代码示例中，重构函数参数类型为 `int` 类型指针，避免出现类型转换：

```
int *Func(int *ptr)
{
    ...
    return ptr;
}

void Caller(int *ptr)
{
    int *intPtr = Func(ptr);
    ...
}
```

G.ARR.06 不要使用变长数组类型

【描述】
在C99中新加入了对变长数组的支持，即数组的长度可以由某个非const变量来定义，变长数组的空间大小直到程序运行时才能确定。
由于要在运行时才能确定数组的大小，因此分配空间的起始地址也是不确定的（例如要在栈上分配两个可变长数组的情况）。这种不确定性会给程序执行带来非常大的风险。

当变长度数组的大小超大时，可能会导致堆栈混乱而引发异常，如果大小为负数或零，那么程序的行为是未定义的。

【反例】

```
void Func(size_t n)
{
    int val[n];       // 不符合
    size_t count = 10;
    int array[count]; // 不符合：可维护性不好

    count = 20;       // 以为修改了array 的大小，但并未改变array是变长数组的本质

    ...
}
```

```
// 不符合: 该声明中的 array 会被调整为指针参数，并不代表 array 的大小是 n
void Func2(int n, int array[n])
{
    ...
}
```

【正例】
解决方案是使用malloc动态分配一段内存，或者在能够明确数组大小时，显式指定其长度。

```
int ReadAndProcess(size_t n)
{
    // 校验n是否合法

    // 解决方案，使用malloc分配大小
    int *array = (int *)malloc(n * sizeof(int));
    // array判空及初始化，此处略
    ...
}
```

```
#define ARRAY_LEN 16

void Func(void)
{
    // 解决方案，明确array的大小，满足此处场景需求。
    int array[ARRAY_LEN];
    ...
}
```

G.ARR.07 声明一个带有外部链接的数组时，必须显式指定它的大小

【描述】
在**声明**具有外部链接的数组时，明确指定其大小会使代码更加清晰可读，有利于加强对数组边界的控制，减少数组读写越界问题。
此规则仍然允许通过初始化列表隐式指定大小的方式来定义一个数组，但在将其声明为一个带有外部链接的数组时，必须显式指定它的大小。
在声明同时也要遵从规则[禁止通过声明的方式引用外部函数接口、变量]

【反例】
如下代码示例中，在头文件中声明了全局数组g_array，但是未显式指定其大小。

```
// in foo.h
extern int g_array[];     // 不符合，没有显式指定数组大小
```

【正例】
如下是正确的代码示例，在头文件中声全局数组`g_array`时，显式指定了数组的大小为`MAX_LEN`。

```
// in foo.h
extern int g_array[MAX_LEN];   // 符合，显式指定了数组大小
```

【正例】
当使用初始化列表隐式指定外部链接数组的大小时，可以定义独立的记录数组长度的常量来使用该数组。

```
//foo.h
const size_t g_privLen;
const char g_priv[];

//foo.c
const char g_priv[] = {'x', 'w', 'r'};
const size_t g_privLen = sizeof(priv) / sizeof(priv[0]);
```

2.11 字符串

G.STR.01 确保字符串有足够的存储空间

【描述】
字符串是一个连续的字符序列，由字符序列中的第一个出现的null字符终止并包含该null字符。拷贝或存储字符串的目标缓冲区必须有足够的空间容纳字符序列包括null结束符，否则可能会导致缓冲区溢出问题。
部分字符串处理函数存在一些隐含的目的缓冲区长度要求，如果未能掌握这些要求，会导致缓冲区写溢出。此类典型函数包括不在C标准库函数中的itoa()/realpath()函数。itoa()/realpath()函数需要在对传入的缓冲区指针位置进行写入操作，但函数并没有提供缓冲区长度。因此，在调用这些函数前，必须提供足够的缓冲区。

【反例】
如下代码示例中，试图将数字转为字符串，但是目标存储空间的预留长度不足。

```
int num = ...
char str[8];
itoa(num, str, 10);  // 10进制整数的最大存储长度是12个字节
```

【正例】
使用安全函数实现整数转换为10进制形式的字符串。

```
int num = ...
char str[16]; // 有时会考虑字节对齐定义冗余的长度，这里选择了16
int ret = sprintf_s(str, sizeof(str), "%d", num);
...  // 处理错误
```

【反例】
如下代码示例中，在对外部数据进行解析并将内容保存到name中，未考虑name的大小。

```
int ProcessMessage(unsigned char *msg, size_t length)
{
    ...
    char name[MAX_NAME];
    size_t i = 0;
    // 必须考虑msg不包含预期的字符'\n'
    while (i < length &&
           msg[i] != '\0' &&
           msg[i] != '\n') {
        name[i] = msg[i];
        i++;
    }
    name[i] = '\0';
    ...
}
```

【正例】
如下代码示例中，在对外部数据进行解析并将内容保存到name中，考虑了name的大小。

```
int ProcessMessage(unsigned char *msg, size_t length)
{
    ...
    char name[MAX_NAME];
    size_t i = 0;
    // 必须考虑msg不包含预期的字符'\n'
    while (i < length &&
           msg[i] != '\0' &&
           msg[i] != '\n' &&
           i < MAX_NAME - 1) { // 使用 MAX_NAME - 1 保留结束符空间
        name[i] = msg[i];
        i++;
    }
    name[i] = '\0';
    ...
}
```

G.STR.02 对字符串进行存储操作，确保字符串有null结束符

【描述】
部分字符串处理函数操作字符串时，将截断超出指定长度的字符串，如strncpy()函数最多复制n个字符到目的缓冲区，如果源字符串长度大于n，则目的缓冲区的内容为n个被复制的字符，null结束符不会被写入到目的缓冲区。使用这类函数时，可能会无意截断导致数据丢失，并在某些情况下会导致软件漏洞。
因此，对字符串进行存储操作，必须确保字符串有null结束符（如使用字符串安全函数生成字符串，或显式对字符数组赋null结束符），否则在后续的调用strlen等操作中，可能会导致内存越界访问漏洞。

【反例】
在如下代码示例中，使用strncpy函数复制字符串时可能会发生截断（发生条件为：`strlen(name) > sizeof(filename) - 1`）。当发生截断时，filename的内容是不完整的，并且缺少'\0'结束符，后续对filename的操作可能会导致软件漏洞：

```
#define FILENAME_LEN 128

char filename[FILENAME_LEN];
strncpy(filename, name, sizeof(filename) - 1);
...
```

【正例】
使用安全函数`strcpy_s`复制字符串，并检查安全函数返回值，如果成功，则确保filename字符串是完整的并且包含'\0'结束符。

```
#define FILENAME_LEN 128

char filename[FILENAME_LEN];
errno_t ret = strcpy_s(filename, sizeof(filename), name);
if (ret != EOK) {
     ... // 处理错误
}
...
```

2.12 断言

断言是一种调试诊断机制，用于验证代码是否符合程序员的预期。程序员在开发期间应该对函数的参数、代码中间执行结果合理地使用断言机制，确保程序的缺陷尽量在测试阶段被发现。

断言可以用于代码中说明各种假定，包括前提条件(preconditions)和后置条件(postconditions)。例如，可以对仅在模块内部使用的函数体内，用断言来声明调用该函数的前提条件，以帮助模块内的调用者正确传入参数。对于模块对外提供的接口函数，由于其实现对外是不可见的，因此不能通过断言来告知调用者需要遵循的约定，也不能通过使用断言来减少对接口函数参数的实际校验。

在调试版本中，断言被触发后，说明程序出现了不应该出现的严重错误，程序会立即提示错误，并终止执行。典型的严重错误如参数在同一模块的上层函数已经校验过，但传递到下层函数后参数不正确，或程序模块内部发送的状态值未定义。

断言必须用宏进行定义，只在调试版本有效，最终发布版本不允许出现assert函数，例如可以按下面的代码实现：

```
#include <assert.h>
#ifdef DEBUG
#define ASSERT(f)  assert(f)
#else
#define ASSERT(f)  ((void)0)
#endif
```

如下的函数VerifyUser，上层调用者会保证传进来的参数是合法的字符串，不可能出现传递非法参数的情况。因此，在该函数的开头，加上4个ASSERT进行校验。

```
bool VerifyUser(const char *userName, const char *password)
{
    ASSERT(userName != NULL);
    ASSERT(strlen(userName) > 0);
    ASSERT(password != NULL);
    ASSERT(strlen(password) > 0);
    ...
}
```

在linux内核中定义ASSERT宏，可以采用如下方式：

```
#ifdef DEBUG
#define ASSERT(f)  BUG_ON(!(f))
#else
#define ASSERT(f)  ((void)0)
#endif
```

G.AST.01 断言必须使用宏定义，且只能在调试版本中生效

【描述】
C语言标准定义断言为用于诊断测试的宏(assert)，因此，断言只能在调试版本中使用。断言被触发后，程序会立即退出，因此严禁在正式发布版本使用断言，请通过编译选项进行控制。

断言触发时虽然能提供少量提示信息，但这样的提示信息通常只对程序员有用，对用户几乎没有价值。程序总是应该优先考虑从错误中恢复，因此，断言应只在调试阶段作为诊断方式生效，其他时候，断言是一种比注释更好的文档说明。例如，断言可以用来声明使用函数的前提条件。

【反例】
如下代码示例在发布版本中使用打印替换断言，生成了实际代码，是不正确的设计。

```
#ifdef DEBUG
#define ASSERT(f)  assert(f)
#else
#define ASSERT(f)  do { \
    if (!(f)) { \
        printf("Error in function=%s, Line=%d\n", __FUNCTION__, __LINE__); \
    } \
} while (0)
#endif
```

G.AST.02 避免在代码中直接使用assert()

【描述】
使用assert()会使代码发布版本与NDEBUG宏发生直接联系。为避免发布版本定义的宏受限于NDEBUG，在代码中不应直接使用assert()。

【反例】

```
int Foo(int *array, size_t size)
{
    /*
     * 违反本条款：当发布版本的编译选项中未指定NDEBUG时，
     * 该代码会被编译到二进制文件中，因此避免在代码中直接使用assert()
     */
    assert(array != NULL);
    ...
}
```

G.AST.03 禁止用断言检测程序在运行期间可能导致的错误，可能发生的错误要用错误处理代码来处理

【描述】
断言主要用于调试期间，在发布版本中应将其关闭。因此，断言应该用于防止不正确的程序员假设，而不能用在发布版本上检查程序运行过程中发生的错误。

断言永远不应用于验证是否存在运行时（与逻辑相对）错误，包括但不限于：

- 无效的用户输入（例如：命令行参数和环境变量）
- 文件错误（例如：打开、读取或写入文件时出错）
- 网络错误（例如：网络协议错误）
- 内存不足的情况（例如：malloc()类似的故障）
- 系统资源耗尽（例如：文件描述符、进程、线程）
- 系统调用错误（例如：执行文件、锁定或解锁互斥锁时出错）
- 无效的权限（例如：文件、内存、用户）

例如，防止缓冲区溢出的代码不能使用断言实现，因为该代码必须编译到发布版本的可执行文件中。如果服务器程序在网运行时由恶意用户触发断言失败，会导致拒绝服务攻击。在这种情况下，更适合使用软故障模式，例如写入日志文件和拒绝请求。

【反例】
以下代码的所有ASSERT的用法都是错误的。例如，错误的使用ASSERT宏来验证内存分配是否成功，因为内存的可用性取决于系统的整体状态，并且在程序运行的任何时候都可能耗尽，所以必须以具有韧性的方式来妥善处理并将程序从内存耗尽中恢复。因此，使用ASSERT宏来验证内存分配是否成功将是不合适的，因为这样做可能导致进程突然终止，从而开启了拒绝服务攻击的可能性。

```
FILE *fp = fopen(path, "r");
ASSERT(fp != NULL);  // 不符合：文件有可能打开失败
char *str = (char *)malloc(MAX_LINE);
ASSERT(str != NULL); // 不符合：内存有可能分配失败
ReadLine(fp, str);
char *p = strstr(str, "age="");
ASSERT(p != NULL);   // 不符合：文件中不一定存在该字符串
char *end = NULL;
long age = strtol(p + 4, &end, 10);
ASSERT(age > 0);     // 不符合：文件内容不一定符合预期
```

【正例】
下面代码演示了如何重构上面的错误代码

```
FILE *fp = fopen(path, "r");
if (fp == NULL) {
    ... // 错误处理
}
char *str = (char *)malloc(MAX_LINE);
if (str == NULL) {
    ... // 错误处理
}
ReadLine(fp, str);
char *p = strstr(str, "age=");
if (p == NULL) {
    ... // 错误处理
}
char *end = NULL;
long age = strtol(p + 4, &end, 10);
if (age <= 0) {
    ... // 错误处理
}
```

G.AST.04 禁止在断言内改变运行环境

【描述】
在程序正式发布阶段，断言不会被编译进去，为了确保调试版和正式版的功能一致性，严禁在断言中使用任何赋值、修改变量、资源操作、内存申请等操作。

例如，以下的断言方式是错误的：

```
ASSERT(p1 = p2);         // p1被修改
ASSERT(i++ > 1000);      // i被修改
ASSERT(close(fd) == 0); // fd被关闭
```

G.AST.05 一个断言只用于检查一个错误

【描述】
为了更加准确地发现错误的位置，每一条断言只校验一个错误。

【反例】
下面的断言同时校验多个错误，在断言触发的时候，无法判断到底是哪一个错误触发了断言：

```
int Foo(int *array, size_t size)
{
    ASSERT(array != NULL && size > 0 && size <= ARRAY_SIZE_MAX);
    ...
}
```

【正例】
应该将每个错误检查分开，可以修改如下：

```
int Foo(int *array, size_t size)
{
    ASSERT(array != NULL);
    ASSERT(size > 0);
    ASSERT(size <= ARRAY_SIZE_MAX);
    ...
}
```

【正例】
如果一个错误检查是由逻辑或组合而成，那么可以写在一个断言中。代码略。

2.13 函数设计

2.13.1 输入校验

P.04 对所有外部数据进行合法性检查

【描述】
外部数据的来源包括但不限于：网络、用户输入、命令行、文件（包括程序的配置文件）、环境变量、用户态数据（对于内核程序）、进程间通信（包括：管道/消息/共享内存/socket/RPC等）、API参数、全局变量。

来自程序外部的数据通常被认为是不可信的，在使用这些数据之前，需要进行合理的检查。
如果不对这些外部数据进行检查，将可能导致不可预期的安全风险。

对来自程序外部的数据要校验处理后才能使用。典型的使用场景包括：

作为数组索引
　将不可信的数据作为数组索引，可能导致超出数组上限，从而造成非法内存访问。
作为内存偏移地址
　将不可信数据作为指针偏移访问内存，可能造成非法内存访问，并可以造成进一步的危害，如任意地址读/写。
作为内存分配的尺寸参数
　例如进行0字节长度分配可能造成非法内存访问，或未限制分配内存大小造成的过度资源消耗。
作为循环条件
　将不可信数据作为循环限定条件，可能会引发缓冲区溢出、内存越界读/写、死循环等问题。
作为除数
　参见除零错误(被零除)。

作为命令行参数

参见“禁止外部可控数据作为进程启动函数的参数”。

作为数据库查询语句的参数

参见“禁止直接使用外部数据拼接SQL命令”。

作为输入/输出格式化字符串

参见“调用格式化输入/输出函数时，禁止format参数由外部可控”。

作为内存拷贝长度

当作为拷贝长度时，可能造成目标缓冲区溢出。

作为文件路径

直接打开不可信路径，可能会导致目录遍历攻击，操作了攻击者无权操作的文件，使得系统被攻击者所控制。

输入校验包括但不局限于：

- 校验数据长度
- 校验数据范围
- 校验数据类型和格式
- 校验输入只包含可接受的字符（“白名单”形式），尤其需要注意一些特殊情况下的特殊字符。

外部数据校验原则

1.信任边界

由于外部数据不可信，因此系统在运行过程中，如果数据传输与处理跨越不同的信任边界，为了防止攻击蔓延，必须对来自信任边界外的其他模块的数据进行合法性校验。

（a）so（或者dll）之间

so或dll作为独立的第三方模块，用于对外导出公共的api函数，供其他模块进行函数调用。so/dll无法确定上层调用者是否传递了合法参数，因此so/dll的公共函数需要检查调用者提供的参数合法性。so/dll应该设计成低耦合、高复用性，尽管有些软件的so/dll当前设计成只在本软件中使用，但仍然应该将不同的so/dll模块视为不同的信任边界。

（b）进程与进程之间

为防止通过高权限进程提权，进程与进程之间的IPC通信（包括单板之间的IPC通信、不同主机间的网络通信），应视为不同信任边界。

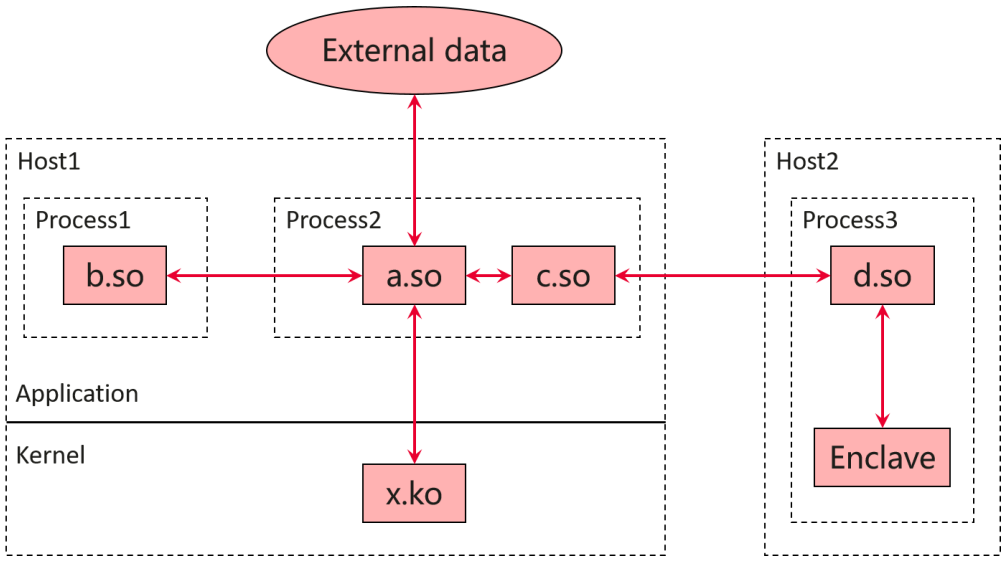
（c）应用层进程与操作系统内核

操作系统内核具有比应用层更高的权限，内核向应用层提供的接口，应该将来自应用层的数据作为不可信数据处理。

（d）可信执行环境内外环境

为防止攻击蔓延至可信执行环境，TEE、SGX等对外提供的接口，应该将来自外部的数据作为不可信数据处理。

下图中红色线条表示数据处理跨越了不同信任面



2.外部数据校验

外部数据进入到本模块后，必须经过合法性校验才能使用。被校验后的合法数据，在本模块内，后续传递到内部其他子函数，不需要重复校验。

下面的例子，函数Foo处理外部数据，由于buffer不一定是'\0'结尾， strlen 的返回值 nameLen 有可能超过 len，导致越界读取数据。本例中采用 strnlen 进行字符串长度计算，随后解析出 name 字符串，在后续的 Foo2 调用中可以直接使用 strlen。

```
void Foo(const unsigned char *buffer, size_t len)
{
    if (buffer == NULL) { // 必须做参数合法性检查
        //错误处理
        ...
    }
    size_t nameLen = strlen((const char *)buffer);    // buffer不一定是'\0'结尾
    char *name = (char *)malloc(nameLen + 1);
    if (name != NULL) {
        errno_t ret = memcpy_s(name, nameLen + 1, buffer, nameLen);
        name[nameLen] = '\0';
    }
    ...
}
```

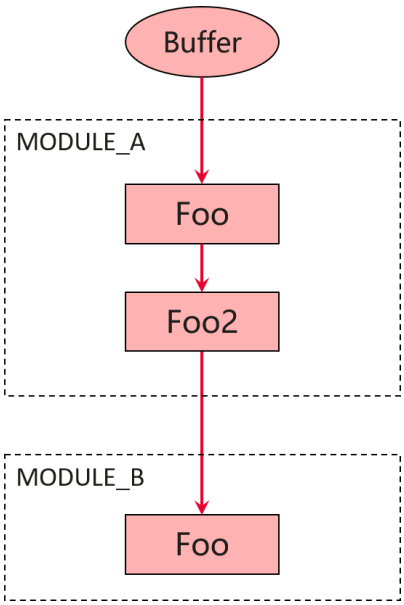
正确的做法，应该是调用 strnlen 避免读越界：

```
void Foo(const unsigned char *buffer, size_t len)
{
    if (buffer == NULL || len >= MAX_BUFFER_LEN) { // 必须做参数合法性检查
        //错误处理
        ...
    }

    // buffer不一定是'\0'结尾
    size_t nameLen = strnlen((const char *)buffer, len);
    char *name = (char *)malloc(nameLen + 1);
    if (name != NULL) {
        memcpy_s(name, nameLen + 1, buffer, nameLen);
        name[nameLen] = '\0';
        Foo2(name);    // foo2内可以直接使用 strlen
    }
    ...
}
```

下面的代码处理外部数据，数据格式如下:

MODULE_A_Foo 和MODULE_A_Foo2是MODULE_A的二个函数，MODULE_A_Foo处理外部数据，将name解析出来，传递给MODULE_A_Foo2处理，MODULE_A_Foo2又调用外部模块MODULE_B的MODULE_B_Foo处理，调用关系如图：




```
// MODULE_A_Foo2 为 MODULE_A 模块的内部函数，约定为由调用者保证参数的合法性
static void MODULE_A_Foo2(const char *name)
{
    // 如果以下的ASSERT触发，表示调用处违反了约定，调用处必须进行修改
    ASSERT(name != NULL);

    size_t nameLen = strlen(name);      //不需要检查name合法性
    MODULE_B_Foo(name);                 //调用MODULE_B中的函数
}

void MODULE_A_Foo(const unsigned char *buffer, size_t len)
{
    if (buffer == NULL || len <= sizeof(int)) { // 必须做参数合法性检查
        // 错误处理
        ...
    }
    int nameLen = *(int *)buffer;
    // nameLen 是不可信数据，必须检查合法性
    if (nameLen <= 0 || (size_t)nameLen > len - sizeof(int)) {
        //错误处理
        ...
    }
    char *name = (char *)malloc(nameLen + 1);
    if (name == NULL) {
        // 内存分配失败，错误处理
        ...
    }
    errno_t err = memcpy_s(name, nameLen + 1, buffer + sizeof(int), nameLen);
    if (err != EOK) {
        // 错误处理
        ...
    }
    name[nameLen] = '\0';    //此时name是一个具有\0结尾的合法字符串
    MODULE_A_Foo2(name);     //调用本模块内内部函数
    ...
}
```

以下是MODULE_B模块中的代码：

```
/*
 * MODULE_B_Foo 为 MODULE_B 模块的公共函数，
 * 其约定为，如果参数name不为NULL，那么必须是一个具有'\0'结尾的合法字符串并且长度大于0
 */
void MODULE_B_Foo(const char *name)
{
    if (name == NULL || name[0] == '\0') { // 必须做参数合法性检查
        // 错误处理
        ...
    }
    size_t nameLen = strlen(name);    // 不需要使用strnlen
    ...
}
```

对于模块A来说， buffer 是外部不可信输入，必须做严格的校验，从 buffer 解析出来的 name，在解析过程中进行了合法性检查，在模块A内部属于合法数据，作为参数传递给内部子函数时不需要再做合法性检查（如果要继续对 name 内容进行解析，那么仍然必须对 name 内容进行校验）。
如果模块A中的 name 继续跨越信任面传递给其他模块（在本例中是直接调用模块B的公共函数，也可以是通过文件、管道、网络等方式），那么对于B模块来说， name 属于不可信数据，必须做合法性检查。

【指南中的关联条款】

- 确保有符号整数运算不溢出
- 确保无符号整数运算不回绕
- 确保除法和余数运算不会导致除零错误(被零除)
- 校验外部数据中整数值的合法性
- 外部数据作为数组索引时必须确保在数组大小范围内
- 将字符串或指针作为函数参数时，在函数体中应检查参数是否为NULL
- 避免使用atoi、atol、atoll、atof函数
- 禁止外部可控数据作为进程启动函数的参数
- 禁止外部可控数据作为dlopen/LoadLibrary等模块加载函数的参数
- 禁止直接使用外部数据拼接SQL命令
- 调用格式化输入/输出函数时，禁止format参数受外部数据控制
- 正确设置安全函数中的destMax参数
- 内存申请前，必须对申请内存大小进行合法性校验
- 外部输入作为内存操作相关函数的复制长度时，需要校验其合法性
- 使用文件路径前必须进行规范化并校验

2.13.2 错误处理

P.05 函数应当合理设计返回值

【描述】
函数应当根据所提供的功能，内部逻辑的复杂程度，以及程序自身的可维护性来设计合适的函数返回值(包括不提供返回值)。

关于返回值的设计策略包括：

1. 如果唯一可能的错误是程序员的错误，那么不要返回错误代码，可以使用断言说明函数的使用前提条件；
2. 如果函数内部涉及多个资源的关闭与释放，而且这些资源的获取可能失败，那么可以考虑使用goto进行函数内的错误处理，释放函数内已获取的资源，并返回错误码；
3. 如果应用程序发生了不可恢复的错误，那么不需要返回错误，可以直接终止程序运行；

4. 如果是公共库的函数中发生了错误，可以调用公共库对外提供的错误回调函数处理错误，并返回错误码；
5. 其他错误，可以返回错误码，或错误标志；
6. 如果函数并不检查错误，而是完成对某个业务逻辑或状态的判断，那么不应返回错误码，而是返回布尔值。

【指南中的关联条款】

- 设计函数时，优先使用返回值而不是输出参数
- 禁止用断言检测程序在运行期间可能导致的错误，可能发生的错误要用错误处理代码来处理
- 将字符串或指针作为函数参数时，在函数体中应检查参数是否为NULL
- 避免使用atoi、atol、atoll、atof函数
- 禁用atexit函数
- 禁止调用kill、TerminateProcess函数直接终止其他进程
- 禁用pthread_exit、ExitThread函数
- 除main函数以外的其他函数，禁止使用exit、ExitProcess函数退出
- 禁用abort函数

G.FUD.01 对象或函数的所有声明必须与定义具有一致的名称和类型限定符

【描述】
在函数或对象的声明和定义中应使用一致的类型和限定符，声明和定义之间的不一致可能存在编程错误，参数名可以提供关于函数接口的有用信息。本规则要求如下：

- 函数声明中的参数应该包含参数名，并且函数定义与其声明中的参数类型以及参数名需要保持一致。
- 如果一个函数声明中没有参数，则在其原型中使用关键字void，否则将导致调用方和被调用方之间的功能接口模糊。
- 同一个对象的声明和定义应保持一致，包括类型、限定符等。

【反例】
如下代码示例中，在func.c中定义了函数，但是在func.h中对该函数的声明存在名称或类型限定符上的不一致，错误的地方已通过注释的方式标识出来。

```
// In func.h
void Func1(const int num);     // 不符合：参数的限定符与函数定义不一致
void Func2(int);               // 不符合：缺少参数名，应修改为void Func2(int num)

// 不符合：参数名不匹配，应修改为 void Func3(int num, int count)
void Func3(int count, int num);

void Func4();                   // 不符合：空参数列表，应修改为void Func4(void)

void (*Fp1)();                  // 不符合：空参数列表，应修改为void (*Fp1)(void)
typedef void (*Fp2)(int);       // 不符合：缺少参数名

// In func.c
int g_a = 0;
int g_array[4] = {0};

void Func1(int num) // 不符合：  参数的限定符与函数声明不一致
{
    ...
}
void Func2(int num)
```

```
{
    ...
}
void Func3(int num, int count)
{
    ...
}
void Func4() // 不符合： 应修改为 void Func4(void)
{
    ...
}

// In caller.c
#include "func.h"
extern short g_a;    // 不符合： 类型不一致，应在func.h中声明
extern int *g_array; // 不符合： 类型不一致，应在func.h中声明

void Caller(void)
{
    Func4(3);        // 不符合： 由于兼容性原因，可以编译通过，但是传入了无用的参数
}
```

【正例】
如下代码示例中，保持声明和定义一致：

```
// In func.h
extern int g_a;
extern int g_array[4];
void Func1(int num);
void Func2(int num);
void Func3(int num, int count);

typedef int Width;
typedef int Height;

void Func4(void)
void (*Fp1)(void);
typedef void (*Fp2)(int num);
typedef void (*Fp3)(int n); // 该规则不要求函数指针中的参数名与其指向函数参数名相同

// In func.c
int g_a = 0;
int g_array[4] = {0};

void Func1(int num)
{
    ...
}
void Func2(int num)
{
    ...
}
void Func3(int num, int count)
{
    ...
}
void Func4(void)
```

```
{
    ...
}
```

G.FUD.02 设计函数时，优先使用返回值而不是输出参数

【描述】
使用返回值而不是输出参数，可以提高可读性，并且通常能够提供相同或更优的性能。

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

G.FUD.03 函数避免使用 void* 类型参数

【描述】
函数参数应尽量避免使用 `void *` 类型，尽量让编译器在编译阶段就检查出类型不匹配的问题。

【反例】
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：

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

typedef struct {
    struct List link;
    int bar;
} BarNode;

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

void MakeTheList(void)
{
    FooNode *foo = NULL;
    BarNode *bar = NULL;
    ...
    FooListAddNode(bar); // 不符合：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。

【正例】
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。

```
// 其他部分同上
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。

G.FUD.04 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

【描述】
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、更安全。

示例：如strncmp 的例子，指向的对象不变化的指针参数声明为const。

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

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。

G.FUD.05 函数要简短

【描述】
函数要简短。复杂过长的函数不利于阅读理解，难以维护。 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，未进行进一步拆分或分层。

产品可从如下维度间接约束函数的尺寸和复杂度：

- 函数行数建议不超过50行（非空非注释）。
- 函数的参数个数。建议不超过5个。
- 函数最大代码块嵌套深度。建议不超过4层。

G.FUD.06 内联函数要尽可能短，避免超过10行

【描述】
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（非空非注释）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。

G.FUD.08 将字符串或指针作为函数参数时，在函数体中应检查参数是否为NULL

【描述】
如果字符串或者指针作为函数参数，为了防止空指针引用错误，在引用前必须确保该参数不为NULL。如果该函数仅用于处理模块内部的可信数据，需要由上层调用者保证该参数不能为NULL，那么在函数开始处可以加断言说明这个参数使用的前提条件。

【正例】
例如下面的代码，因为int *p有可能为NULL，因此在使用前需要进行判断。

```
#define MAX_COUNT ...
int Func(int *p, size_t count)
{
    if (p == NULL || count == 0 || count > MAX_COUNT) {
        ... // 错误处理
    }
    int c = p[0];
    ...
}
int Caller(void)
{
    int *arr = ...
    size_t count = ...
    Func(arr, count);
    ...
}
```

下面的代码，由于p的合法性由调用者保证，对于 Func() 函数，不可能出现p为NULL的情况，因此加上断言进行校验。

```
int Func(int *p, size_t count)
{
    ASSERT(p != NULL); // 由调用者保证p不为空
    ASSERT(count > 0);
    ASSERT(count <= MAX_COUNT);
    int c = p[0];
    ...
}

int Caller(void)
{
    int *arr = ...
    size_t count = ...
    ...
    if (arr != NULL && count > 0 && count <= MAX_COUNT) {
        Func(arr, count);
    }
    ...
}
```

G.FUD.09 避免修改函数参数的值

【描述】
函数设计时应充分考虑其参数的用途，如果不需要修改的函数参数被不合理地修改了，其执行结果可能与程序员的期望不一致，会给代码开发、维护带来潜在的风险。

【反例】
如下代码示例中，由于不合理地修改了参数的值，在`if(input > threshold)`条件判断时，改变了程序的预期，导致逻辑错误。

```
void Func(int input, int *output)
{
    ...  // 包含input的合法性校验，确保不会在引起函数中的整数运算溢出问题
    input += Add(input);      // 不符合：修改参数的值，给后边的代码开发、维护造成迷惑
    threshold *= Multiplier(input);
    ...
```

```
    // 该条件语句本意是使用原始入参input值，但input在前面已被修改，导致程序执行时与预期不一致
    if (input > threshold) {
        DoExtraOperation();
        ...
    }

    *output = input;          // input 已被修改，给阅读、维护代码造成迷惑
}
```

【正例】
如下代码示例中，使用单独的局部变量作为工作变量。

```
void Func(int input, int *output)
{
    ...  // 包含input的合法性校验，确保不会在引起函数中的整数运算溢出问题
    int workVar = input;      // 符合：使用局部变量代替

    workVar += Add(workVar);
    threshold *= Multiplier(workVar);
    ...
    if (input > threshold) { // 原始入参值没变，和期望一致
        DoExtraOperation();
        ...
    }

    *output = workVar;        // 符合，没有修改函数参数的值
}
```

G.FUD.10 函数只在一个文件内使用时应当使用static修饰符

【描述】
函数如果只在一个文件内使用，应当使用static修饰符。

如果函数声明的范围比需要的范围大，则会降低代码的可读性，并可能被意外引用而产生预期之外的错误。

【反例】
如下代码示例中，Foo()函数仅在Goo()函数内部调用。
对于C语言，默认情况下，该Foo()函数等同于声明了extern，这意味着函数位于全局符号表中，并且可以从其他.c文件中的函数调用。

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

int Goo(int value)
{
    int result = Foo(value);
    ...
}
```

【正例】
如下代码示例中，该函数Foo()使用内部链接声明static。

这种做法将函数声明的范围限制在当前文件中，并防止该函数包含在外部符号表中。它还限制了全局命名空间中的混乱情况，并防止函数被其他文件调用。

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

int Goo(int value)
{
    int result = Foo(value);
    ...
}
```

2.14 函数使用

2.14.1 函数返回值

G.FUU.01 处理函数的返回值

【描述】
函数通常通过返回值返回数据或执行结果，调用者应该在函数调用之后，对返回数据、执行结果进行有效、正确的合法性检查和处理。

如果调用者有意不处理返回值，在经过充分考虑之后，可用`(void)`显式忽略掉。

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类型。

【反例】

```
char *p = (char *)malloc(SOME_SIZE);    // 不符合：未检查返回值 p 的合法性
memset_s(p, SOME_SIZE, 0, SOME_SIZE);   // 不符合：memset_s 返回值未处理
```

【正例】

```
char *p = (char *)malloc(SOME_SIZE);
if (p == NULL) {    // 符合：对返回数据进行合法性检查
    ...
    return ...;
}
// 符合：此处符合可省略memset_s返回值检查的例外场景，可以显式忽略掉返回值的检查
(void)memset_s(p, SOME_SIZE, 0, SOME_SIZE);
```

2.14.2 格式化输入/输出函数

G.FUU.02 调用格式化输入/输出函数时，禁止format参数受外部数据控制

【描述】
调用格式化函数时，如果format参数由外部数据提供，或由外部数据拼接而来，会造成格式化字符串漏洞。
以格式化输出函数为例，当其format参数外部可控时，攻击者可以使用%n转换符向指定地址写入一个整数值、使用%x或%d转换符查看栈或寄存器内容、使用%s转换符造成进程崩溃等。

常见格式化函数有：

- 格式化输出函数: xxxprintf;
- 格式化输入函数: xxxscanf;
- 格式化错误消息函数: err(), verr(), errx(), verrx(), warn(), vwarn(), warnx(), vwarnx(), error(), error_at_line();
- 格式化日志函数: syslog(), vsyslog().

【反例】
如下代码示例中，使用printf()函数直接打印外部数据，可能出现格式化字符串漏洞。

```
void Foo(void)
{
    const char *msg = GetMsg();
    ...
    printf(msg); // 存在格式化字符串漏洞
}
```

【正例】
下面是推荐做法，使用%s转换符打印外部数据，避免格式化字符串漏洞。

```
void Foo(void)
{
    const char *msg = GetMsg();
    ...
    printf("%s", msg); // 这里没有格式化字符串漏洞
}
```

G.FUU.03 调用格式化输入/输出函数时，使用有效的格式字符串

【描述】
格式化输入/输出函数（如scanf()/printf()及相关函数）在format字符串控制下进行转换、格式化、打印其实参。
使用这些函数时，需要确保格式串是合法有效的，并且格式串与相应的实参类型是严格匹配的，否则会使程序产生非预期行为。

【反例】
如下代码示例中，格式化输入一个整数到macAddr变量中，但是macAddr为unsigned char类型，而%x对应的是int类型参数，函数执行完成后会发生写越界。

```
unsigned char macAddr[6];
...
// macStr中的数据格式为 e2:42:a4:52:1e:33
int ret = sscanf_s(macStr, "%x:%x:%x:%x:%x:%x\n",
                   &macAddr[0], &macAddr[1],
                   &macAddr[2], &macAddr[3],
                   &macAddr[4], &macAddr[5]);
...
```

【正例】
如下代码中，使用%hhd确保格式串与相应的实参类型严格匹配。

```
unsigned char macAddr[6];
...
// macStr中的数据格式为 e2:42:ca4:52:1e:33
int ret = sscanf_s(macStr, "%hhx:%hhx:%hhx:%hhx:%hhx:%hhx\n",
                   &macAddr[0], &macAddr[1],
                   &macAddr[2], &macAddr[3],
                   &macAddr[4], &macAddr[5]);
...
```

2.14.3 退出类函数

G.FUU.04 禁用atexit函数

【描述】
atexit函数使用有严格限制：在一个程序中最多只能注册32个例程；例程必须通过return返回，不允许调用退出函数（如exit()）或调用longjmp()等方式返回，否则例程可能得不到正确执行，还会造成程序产生未定义行为。同时对于所有的退出处理程序来说，程序员应该主动清理不再使用的资源，而不应该在最终程序退出后通过使用atexit函数事先注册的例程被动地清理资源。

【反例】
在如下代码示例中，ProcessExit()函数由atexit()事先注册，在程序终止时执行必要的资源清理操作。但是如果g_someResource条件为真，则exit()被第二次调用，造成程序产生未定义行为。

```
...
int g_someResource = 1;

void ProcessExit(void)
{
    if (g_someResource == 0) {
        ... // g_someResource，在这里清理程序资源，代码省略

        exit(0); // 不符合：在本例程中调用exit，导致程序产生未定义行为
    }

    return;
}

int main(void)
{
    // 不符合：注册atexit()例程ProcessExit()，清理资源
    if (atexit(ProcessExit) != 0) {
        ... // 错误处理
    }

    ...

    // main函数退出
    return 0;
}
```

【正例】
重构代码，禁止使用atexit()函数

```
...
void ProcessExit(void)
```

```
{
    if (g_someResource == 0) {
        ... // g_someResource，在这里清理程序资源，代码省略
    }

    return;
}

int main(void)
{
    ...

    // main函数退出
    ProcessExit(); // 符合：主动调用资源清理函数
    return 0;
}
```

【例外】

作为服务维测监控功能，为定位程序异常退出原因的模块，可以作为例外使用atexit()函数。

G.FUU.05 禁止调用kill、TerminateProcess函数直接终止其他进程

【描述】

调用kill、TerminateProcess等函数直接强行终止其他进程(如kill -SIGKILL，kill -SIGSTOP)，会导致被终止的进程中的资源得不到清理。

进程终止时，进程特有的资源将由系统自动释放，而其他资源(尤其是不属于进程的系统资源)在程序退出前难以得到有效地清理。

对于进程/线程间通信，应该主动发送一个停止命令，通知对方安全退出。接收到停止命令的进程应尽快完成进程资源和不再使用的系统资源的清理和释放。

当发送给对方进程/线程退出信号后，在等待一定时间内如果对方仍然未退出，可以调用kill、TerminateProcess函数强行终止目标进程。

【反例】

```
if (isFatalStatus) {

    ...

    kill(pid, SIGKILL); // 不符合：直接调用kill强行结束目标进程
}
```

【正例】

正确的做法是通知对方进程停止，在等待一定时间内如果对方仍然未退出，再强行终止目标进程。

```
if (isFatalStatus) {

    ...

    kill(pid, SIGUSR1); // 符合：目标进程将SIGUSR1定义为停止命令

    ...

    if (WaitForRemoteProcessExit() == TIME_OUT) {
        kill(pid, SIGKILL); // 目标进程在限定时间内仍然未退出，强行结束目标进程
    }
}
```

G.FUU.06 禁用pthread_exit、ExitThread函数

【描述】

严禁在线程内主动终止自身线程，线程函数在执行完毕后会自动、安全地退出。主动终止自身线程的操作，不仅导致代码复用性变差，同时容易导致资源泄漏错误。

pthread_exit()函数将终止调用它的线程，线程终止时不释放任何应用程序可见的进程资源，包括但不限于互斥锁和文件描述符，也不执行任何进程级别的清理操作。

windows环境下ExitThread用于使当前正在运行的线程正常且干净地停止。运行该函数可能导致主线程退出，而其他线程仍然在执行。停止线程的正确方法是发出干净的退出信号，然后允许线程自行退出，然后再允许主线程退出。

G.FUU.07 除main函数以外的其他函数，禁止使用exit、ExitProcess函数退出

【描述】

除了main函数以外，禁止任何地方调用exit、ExitProcess函数退出进程。直接退出进程可能会导致代码的复用性降低，某些资源（尤其是不属于进程的系统资源）在程序退出前难以得到有效地清理。

处理程序退出时必须通过不断返回上一层函数来终止运行。对于所有退出处理程序来说，能够正确执行它们的清理操作是很重要的，在安全性上可能很关键。

【反例】

以下代码加载文件，加载过程中如果出错，直接调用exit退出，可能会造成前面的资源未清理。

```
void LoadFile(const char *filePath)
{
    ASSERT(filePath != NULL);

    FILE *fp = fopen(filePath, "rt");
    if (fp == NULL) {
        exit(0); // 不符合
    }

    ...
}
```

【正例】

正确的做法应该通过错误值传递机制返回上一层调用者，例如：

```
bool LoadFile(const char *filePath)
{
    ASSERT(filePath != NULL);
```

```
    bool ret = false;
    FILE *fp = fopen(filePath, "rt");
    if (fp == NULL) {
        ... // 错误处理
        return ret;
    }
    ...

    return ret;
}
```

G.FUU.08 禁用abort函数

【描述】
abort()将会导致程序立即退出，资源得不到清理。因为在使用abort()退出程序时，不能保证输入/输出流已经被清空，文件已经正确关闭，或者临时文件已经删除。

【例外】

只有发生致命错误，程序无法继续执行的时候，在错误处理函数中使用abort退出程序，例如：

```
void FatalError(int sig)
{
    abort();
}

int main(int argc, char *argv[])
{
    signal(SIGSEGV, FatalError);

    ...
}
```

2.14.4 内存类函数

G.FUU.09 禁止使用realloc()函数

【描述】
realloc()是一个非常特殊的函数，原型如下：

```
void *realloc(void *ptr, size_t size);
```

随着参数的不同，其行为也是不同：

- 当ptr不为NULL，且size不为0时，该函数会重新调整内存大小，并将新的内存指针返回，并保证最小的size的内容不变；
- 参数ptr为NULL，但size不为0，那么其行为等同于malloc(size)；
- 参数size为0，则realloc的行为等同于free(ptr)。

由此可见，一个简单的C函数，却被赋予了3种行为，这不是一个设计良好的函数。虽然在编程中提供了一些便利性，如果认识不足，使用不当，是却极易引发各种bug。

【反例】
如下代码示例中，使用realloc不当导致内存泄漏。
代码中希望对ptr的空间进行扩充，当realloc()分配失败的时候，会返回NULL。但是参数中的ptr的内存是没有被释放的，如果直接将realloc()的返回值赋给ptr，那么ptr原来指向的内存就会丢失，造成内存泄

漏。

```
// 当realloc()分配内存失败时会返回NULL，导致内存泄漏
char *ptr = (char *)realloc(ptr, NEW_SIZE);
if (ptr == NULL) {
    .. // 错误处理
}
```

【正例】
使用malloc()函数代替realloc()函数。

```
// 使用malloc()函数代替realloc()函数
char *newPtr = (char *)malloc(NEW_SIZE);
if (newPtr == NULL) {
    ... // 错误处理
}

errno_t ret = memcpy_s(newPtr, NEW_SIZE, oldPtr, oldSize);

... // 校验ret，确保安全函数执行成功

... // 返回前，释放oldPtr
```

G.FUU.10 禁止使用alloca()函数申请栈上内存

【描述】
POSIX和C99均未定义alloca()的行为，在有些平台下不支持该函数，使用alloca会降低程序的兼容性和可移植性，该函数在栈帧里申请内存，申请的大小很可能超过栈的边界，影响后续的代码执行。

请使用malloc从堆中动态分配内存。

2.14.6 其他函数

G.FUU.16 禁止外部可控数据作为进程启动函数的参数

【描述】
本条款中进程启动函数包括system、popen、execl、execlp、execle、execv、execvp等。
system()、popen()等函数会创建一个新的进程，如果外部可控数据作为这些函数的参数，会导致注入漏洞。
使用execl()等函数执行新进程时，如果使用shell启动的新进程，则同样存在命令注入风险。
使用execlp()、execvp()、execvpe()函数依赖于系统的环境变量PATH来搜索程序路径，使用它们时应充分考虑外部环境变量的风险，或避免使用这些函数。
因此，总是优先考虑使用C标准函数实现需要的功能。如果确实需要使用这些函数，请使用白名单机制确保这些函数的参数不受任何外来数据的影响。

【反例】
如下代码示例中，使用 system() 函数执行 cmd 命令串来自外部，攻击者可以执行任意命令：

```
char *cmd = GetCmdFromRemote();
if (cmd == NULL) {
    ... // 处理错误
}
system(cmd);
```

如下代码示例中，使用 system() 函数执行 cmd 命令串的一部分来自外部，攻击者可能输入 'some dir;reboot'字符串，造成系统重启：

```
char cmd[MAX_LEN];
int ret = 0;
char *name = GetDirNameFromRemote();
if (name == NULL) {
    ... // 处理错误
}
...
ret = sprintf_s(cmd, sizeof(cmd), "ls %s", name)
...
system(cmd);
```

使用exec系列函数来避免命令注入时，注意exec系列函数中的path、file参数禁止使用命令解析器(如/bin/sh)。

```
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const char *path, const char *arg, ...);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[], char *const envp[]);
```

例如，禁止如下使用方式：

```
char *cmd = GetDirNameFromRemote();
execl("/bin/sh", "sh", "-c", cmd, NULL);
```

【正例】
对输入的文件名基于合理的白名单检查，避免命令注入。

```
char *cmd = GetCmdFromRemote();
if (cmd == NULL) {
    ... // 处理错误
}

// 使用白名单检查命令是否合法，仅允许"some_tool_a", "some_tool_b"命令，外部无法随意控制
if (!IsValidCmd(cmd)) {
    ... // 处理错误
}
system(cmd);
...
```

G.FUU.17 禁止外部可控数据作为dlopen/LoadLibrary等模块加载函数的参数

【描述】
这些函数会加载外部模块，如果外部可控数据作为这些函数的参数，有可能会加载攻击者事先预制的模块。

如果要使用这些函数，可以采用如下措施之一：

- 使用白名单机制，确保这些函数的参数不受任何外来数据的影响；
- 使用数字签名机制保护要加载的模块，充分保证其完整性；

- 在设备本地加载的动态库通过权限与访问控制措施保证了本身安全性后，通过特定目录自动被程序加载；
- 在设备本地的配置文件通过权限与访问控制措施保证了本身安全性后，自动加载配置文件中指定的动态库。

【反例】
以下代码从外部获取数据后直接作为LoadLibrary函数的参数，有可能导致程序被植入木马。

```
char *msg = GetMsgFromRemote();
LoadLibrary(msg);
```

G.FUU.18 禁止直接使用外部数据拼接SQL命令

【描述】
SQL注入是指SQL查询被恶意更改成一个与程序预期完全不同的查询。执行更改后的查询可能会导致信息泄露或者数据被篡改。而SQL注入的根源就是使用外部数据来拼接SQL语句。C/C++语言中常见的使用外部数据拼接SQL语句的场景有（包括但不局限于）：

- 连接MySQL时调用mysql_query()，Execute()时的入参
- 连接SQL Server时调用db-library驱动的dbsqlexec()的入参
- 调用ODBC驱动的SQLprepare()连接数据库时的SQL语句的参数
- C++程序调用OTL类库中的otl_stream()，otl_column_desc()时的入参
- C++程序连接Oracle数据库时调用ExecuteWithResSQL()的入参

防止SQL注入的方法主要有以下几种：

- 参数化查询（通常也叫作预处理语句）：参数化查询是一种简单有效的防止SQL注入的查询方式，应该被优先考虑使用。支持的数据库有MySQL，Oracle（OCI）。
- 参数化查询（通过ODBC驱动）：支持ODBC驱动参数化查询的数据库有Oracle、SQLServer、PostgreSQL和GaussDB。
- 对外部数据进行校验（对于每个引入的外部数据推荐“白名单”校验）。
- 对外部数据中的SQL特殊字符进行转义。

【反例】
下列代码拼接用户输入，没有进行输入检查，存在SQL注入风险：

```
char name[NAME_MAX];
char sqlStatements[SQL_CMD_MAX];
int ret = GetUserInput(name, NAME_MAX);
...
ret = sprintf_s(sqlStatements,
                SQL_CMD_MAX,
                "SELECT childinfo FROM children WHERE name= ‘%s’",
                name );
...
ret = mysql_query(&myConnection, sqlStatements);
...
```

【正例】
使用预处理语句进行参数化查询可以防御SQL注入攻击：

```
char name[NAME_MAX];
...
MYSQL_STMT *stmt = mysql_stmt_init(myConnection);
char *query = "SELECT childinfo FROM children WHERE name= ?";
```

```
if (mysql_stmt_prepare(stmt, query, strlen(query))) {
    ...
}
int ret = GetUserInput(name, NAME_MAX);
...
MYSQL_BIND params[1];
(void)memset_s(params, sizeof(params), 0, sizeof(params));
...
params[0].bufferType = MYSQL_TYPE_STRING;
params[0].buffer = (char *)name;
params[0].bufferLength = strlen(name);
params[0].isNull = 0;

bool isCompleted = mysql_stmt_bind_param(stmt, params);
...
ret = mysql_stmt_execute(stmt);
...
```

G.FUU.19 禁止在信号处理例程中调用非异步安全函数

【描述】
信号处理例程应尽可能简化。在信号处理例程中如果调用非异步安全函数，可能会导致程序的执行不符合预期的结果。

【反例】
如下代码示例中，信号处理函数中调用了非异步安全函数fprintf()函数。

```
FILE *g_logFile = NULL;

static void SigHandler(int num)
{
    if (g_logFile != NULL) {
        fprintf(g_logFile, ...);
    }
}

int main(int argc, char **argv)
{
    ...

    g_logFile = ...
    ...
    if (signal(SIGUSR1, SigHandler) == SIG_ERR) {
        ... // 错误处理
    }
    ...
    return 0;
}
```

G.FUU.20 使用库函数时避免竞争条件

【描述】
在编写含有并发操作的程序时，调用C语言标准库函数需避免竞争条件。因此，使用库函数前需仔细阅读函数说明文档，避免因并发操作而产生竞争条件问题。

【反例】
以下示例是期望通过getenv函数取到TABLE_WX_ENV和TABLE_BD_ENV两个环境变量的值，再判断这两个值是否相同。
但是由于getenv()是不可重入的，在第二次调用该函数时，可能会覆盖上一次取到的字符串内容，所以即使两个环境变量是不同的，但是最终两者wxEnv和bdEnv在比较时可能相同。

```
void Foo(void)
{
    char *wxEnv = NULL;
    char *bdEnv = NULL;

    wxEnv = getenv("TABLE_WX_ENV");
    if (wxEnv == NULL) {
        ... // 错误处理
        return;
    }

    bdEnv = getenv("TABLE_BD_ENV");
    if (bdEnv == NULL) {
        ... // 错误处理
        return;
    }

    if (strcmp(wxEnv, bdEnv) == 0) {
        printf("They are the same.\n");
    } else {
        printf("They are NOT the same.\n");
    }
}
```

【正例】
如下代码示例中，先将取到的环境变量保存起来，然后再比较。

```
// 函数是将srcEnv复制一份。成功返回复制之后的首地址，失败返回NULL
char *TransEnv(const char *srcEnv)
{
    ... // 具体实现略。strcpy_s...
}

void Function(void)
{
    char *wxEnv = NULL;
    char *bdEnv = NULL;

    const char *envVal = getenv("TABLE_WX_ENV");
    if (envVal != NULL) {
        wxEnv = TransEnv(envVal);
        ... // 判断TransEnv函数返回值并正确处理异常
    } else {
        ... // 错误处理
        return;
    }

    envVal = getenv("TABLE_BD_ENV");
    if (envVal != NULL) {
        bdEnv = TransEnv(envVal);
```

```
        ... // 判断TransEnv函数返回值并正确处理异常
    } else {
        ... // 错误处理
        return;
    }

    // 比较两个副本
    if (strcmp(wxEnv, bdEnv) == 0) {
        printf("They are the same.\n");
    } else {
        printf("They are NOT the same.\n");
    }

    ... // 退出前清理资源
}
```

G.FUU.22 避免使用atoi、atol、atoll、atof函数

【描述】
字符串中可能不含数字、含有其他字符或含有超出表示范围的数字，因此，将字符串转换为数值时，必须检测并解决这些错误情况。

如果使用C标准库函数做字符串到数值的转换，应当使用 strtol()、strtoll()、strtod()等函数，代替 atoi()、atol()、atoll()、atof()等函数，主要原因如下：

- atoi()、atol()、atoll()、atof()系列函数。当字符串转换的结果无法表示成相应的数值时，会导致程序产生未定义行为，并且未提供足够的出错信息；
- strtol()、strtoll()、strtod()系列函数。这些函数能够完成与上一组函数相同的功能，函数行为是确定的，并且提供了更多的错误检测能力。

【反例】
如下示例代码中，atol()函数无法判断转换过程中是否存在错误：

```
char buf[BUF_LEN];
if (fgets(buf, BUF_LEN, stdin) == NULL) {
    ... // 错误处理
}

/*
 * 当字符串不是以数字开始时，函数返回 0，并且不会设置errno值，调用者无法判断是否存在错误
 * 当字符串中间存在非数字字符时，函数返回已转换的数字，调用者无法判断是否完全转换
 * 当字符串中的数字超出目标类型表示范围时，程序产生未定义行为，调用者无法判断是否存在错误
 */
long result = atol(buf);
...
```

【正例】
如下示例代码中，使用strtol()函数代替atol()函数转换字符串到整数，并校验是否存在错误：

```
#define NUMBER_BASE 10

char buf[BUF_LEN];
char *endPtr = buf;
if (fgets(buf, BUF_LEN, stdin) == NULL) {
    ... // 错误处理
}
```

```
errno = 0;
long result = strtol(buf, &endPtr, NUMBER_BASE);

/*
 * 调用者可以通过检查 endPtr 的值和内容，来判断是否转换完全，
 * 可以通过 errno 是否为 ERANGE来判断是否超出整数表示范围
 */
if (endPtr == buf || *endPtr != '\0' || errno == ERANGE) {
    ... // 错误处理
}
...
```

需要注意的是，实际业务使用`strto*`函数时，应根据自身的业务设计需求设计错误检查的代码。

2.15 内存

G.MEM.01 内存申请前，必须对申请内存大小进行合法性校验

【描述】
当申请内存大小由程序外部输入时，内存申请前，要求对申请内存大小进行合法性校验，防止申请0长度内存，或过多地、非法地申请内存。
当申请0长度内存时，内存分配函数的行为是由实现定义的，通常返回非空指针，读写该内存可能造成程序异常。当申请内存的数值过大时（可能一次就申请了非常大的超预期的内存；也可能循环中多次申请内存），可能会造成非预期的资源耗尽。

大小不正确的参数、不当的范围检查、整数溢出或者截断都可能造成实际分配的缓冲区不符合预期。如果申请内存受攻击者控制，还可能会发生缓冲区溢出等安全问题。

【反例】
如下代码示例中，动态分配size大小的内存，未校验size是否为0或者超出上限值。

```
// 这里的size在传入DoSomething()函数之前还未做过合法性校验
int DoSomething(size_t size)
{
    ...
    char *buffer = (char *)malloc(size); // 本函数内，size使用前未做检查
    if (buffer == NULL) {
        ... // 处理 malloc() 错误
    }
    ...  // 业务处理
    free(buffer);
}
```

【正例】
如下代码示例中，动态分配size大小的内存前，进行了符合程序需要的合法性校验。

```
// 这里的size在传入DoSomething()函数之前还未做过合法性校验
int DoSomething(size_t size)
{
    ...
    /*
     * 本函数内，size使用前做合法性检查，
     * FOO_MAX_LEN被定义为符合程序设计预期的最大内存空间
     */
    if (size == 0 || size > FOO_MAX_LEN) {
```

```
        ... // 错误处理
    }
    char *buffer = (char *)malloc(size); // 本函数内，size使用前未做检查
    if (buffer == NULL) {
        ... // 处理 malloc() 错误
    }
    ...  // 业务处理
    free(buffer);
}
```

G.MEM.02 内存分配后必须判断是否成功

【描述】
内存分配一旦失败，那么后续的操作会导致程序产生未定义行为的风险。比如malloc申请失败返回了空指针，对空指针的解引用会导致程序产生未定义行为。

【反例】
如下代码示例中，调用malloc分配内存之后，没有判断是否成功，直接引用了p。如果malloc失败，它将返回一个空指针并传递给p。当如下代码在内存拷贝中解引用了该空指针p时，程序会出现未定义行为。

```
struct tm *MakeTm(int year, int mon, int day, int hour,
                  int min, int sec)
{
    struct tm *tmb = (struct tm *)malloc(sizeof(*tmb));
    tmb->year = year;
    ...
    return tmb;
}
```

【正例】
如下代码示例中，在malloc分配内存之后，立即判断其是否成功，消除了上述的风险。

```
struct tm *MakeTm(int year, int mon, int day, int hour,
                  int min, int sec)
{
    struct tm *tmb = (struct tm *)malloc(sizeof(*tmb));
    if (tmb == NULL) {
        ... // 错误处理
    }
    tmb->year = year;
    ...
    return tmb;
}
```

G.MEM.03 外部输入作为内存操作相关函数的复制长度时，需要校验其合法性

【描述】
将数据复制到容量不足以容纳该数据的内存中会导致缓冲区溢出。为了防止此类错误，必须根据目标容量的大小限制被复制的数据大小，或者必须确保目标容量足够大以容纳要复制的数据。

【反例】
外部输入的数据不一定会直接作为内存复制长度使用，还可能会间接参与内存复制操作。
如下代码示例中，inputTable->count来自设备外部报文，虽然没有直接作为内存复制长度使用，而是作为for循环体的上限使用，间接参与了内存复制操作。由于没有校验其大小，可造成缓冲区溢出：

```
typedef struct {
    size_t count;
    int val[MAX_NUMBERS];
} ValueTable;

ValueTable *ValueTableDup(const ValueTable *inputTable)
{
    ValueTable *OutputTable = ... // 分配内存
    ...
    for (size_t i = 0; i < inputTable->count; i++) {
        OutputTable ->val[i] = inputTable->val[i];
    }
    ...
}
```

【正例】
如下代码示例中，对inputTable->count做了校验。

```
typedef struct {
    size_t count;
    int val[MAX_NUMBERS];
} ValueTable;

ValueTable *ValueTableDup(const ValueTable *inputTable)
{
    ValueTable *OutputTable = ... // 分配内存
    ...
    /*
     * 根据应用场景，对来自外部报文的循环长度inputTable->count
     * 与OutputTable->val数组大小做校验，避免造成缓冲区溢出
     */
    if (inputTable->count >
        sizeof(OutputTable->val) / sizeof(OutputTable->val[0]) {
        return NULL;
    }
    for (size_t i = 0; i < inputTable->count; i++) {
        OutputTable ->val[i] = inputTable->val[i];
    }
    ...
}
```

G.MEM.04 内存中的敏感信息使用完毕后立即清0

【描述】
内存中的口令、密钥等敏感信息使用完毕后立即清0，避免被攻击者获取或者无意间泄露给非授权用户。

G.MEM.05 不要访问已释放的内存

【描述】
如果指针所指向的内存空间已被释放，那么再次使用这些指针值会导致程序产生未定义行为（如解引用已释放内存的指针，将这些指针作为free()函数的参数再次进行释放等）。
再次使用已释放内存的指针，可能因访问无效内存造成程序崩溃，在精心构造的条件下，还可能因破坏内存的管理机制造成恶意代码执行的问题。

【反例】
如下代码，data释放以后，由于程序逻辑设计疏忽，在后面的代码中再次使用了其成员ctx，造成错误。

```
...
struct MemCb *data = NULL;
...  // 初始化data结构
if (DealingData(data) == NULL) { // 处理data数据
    DBG_LOG("Dealing Data Error");
    free(data);
}
...
ret = ProcessMemCbCtx(data->ctx); // 错误引用了已释放内存data中的成员
...
```

【正例】
如下代码在指针释放后，将其重置为空值，并在使用前进行了判空，避免了使用已释放内存的问题。

```
...
struct MemCb *data = NULL;
...  // 分配并初始化data结构
if (DealingData(data) == NULL) { // 处理data数据，如果失败，则释放data内存
    DBG_LOG("Dealing Data Error");
    free(data);
    data = NULL;  // 释放data内存后，将NULL赋给指针
}
...
if (data != NULL) { // 使用data及其成员前，先判断指针是否为NULL
    ret = ProcessMemCbCtx(data->ctx);
    ...
}
...
```

特别需要注意的是，如果在释放data前，将data拷贝给了其他的指针，那么data释放后，data的副本指针也不能被再次使用。

2.16 文件

自UNIX开始引入文件等概念后，输入输出操作可以独立于具体设备进行。输入输出模型的细节被隐藏在文件结构之下。因此，不应当访问FILE指针的成员。此外，C语言标准还禁止复制文件对象。

在并发系统上访问文件可能存在竞争问题。访问文件时，应尽量避免条件竞争。

G.FIL.01 创建文件时必须显式指定合适的文件访问权限

【描述】
创建文件时，如果不显式指定合适访问权限，可能会让非授权用户访问该文件，造成信息泄露，文件数据被篡改，文件中被注入恶意代码等风险。

【反例】
如下代码中，设置文件的访问权限不合适，过于宽松，所有用户均可以访问该文件，并且文件可执行，存在安全隐患。

```
...
// 访问权限过于宽松
int fd = open(filename, O_WRONLY | O_CREAT | O_EXCL, S_IRWXU | S_IRWXG |
S_IRWXO);
if (fd == -1) {
    ... // 错误处理
}
...
```

【正例】
如下代码中，根据文件实际的应用情况设置访问权限，只允许属主可以读写该文件。

```
...
// 此处根据文件实际需要，指定其访问权限
int fd = open(filename, O_WRONLY | O_CREAT | O_EXCL, S_IWUSR | S_IRUSR);
if (fd == -1) {
    ... // 错误处理
}
...
```

G.FIL.02 使用文件路径前必须进行规范化并校验

【描述】
当文件路径来自外部数据时，必须对其做合法性校验，如果不校验，可能造成系统文件的被任意访问。但是禁止直接对其进行校验，正确做法是在校验之前必须对其进行路径规范化处理。这是因为同一个文件可以通过多种形式的路径来描述和引用，如果不进行规范化处理，则可能会绕过校验，例如：路径既可以是绝对路径，也可以是相对路径或者是符号链接。

因为规范化机制在不同的操作系统和文件系统之间可能有所不同，所以最好使用符合当前系统特性的规范化机制。例如：在linux下，使用realpath函数，在windows下，使用PathCanonicalize函数进行文件路径的规范化。

一个简单的案例说明如下：

> 当文件路径来自外部数据时，需要先将文件路径规范化，如果没有作规范化处理，攻击者就有机会通过恶意构造文件路径进行文件的越权访问。
> 例如，攻击者可以构造“../../../etc/passwd”的方式进行任意文件访问。

【例外】
运行于控制台的命令行程序，通过控制台手工输入文件路径，可以作为本条款例外。

```
int main(int argc, char **argv)
{
    if (argc >= 2 && argv[1] != NULL) {
        int fd = open(argv[1], O_RDONLY);
        ...
    }
    ...
    return 0;
}
```

G.FIL.03 不要在共享目录中创建临时文件

【描述】
共享目录是指其它非授权用户可以访问的目录。程序的临时文件应当是程序自身独享的，任何将自身临时文件置于共享目录的做法，将导致其他非授权用户获得该程序的额外信息，产生信息泄露等风险。因此，不要在任何共享目录创建仅由程序自身使用的临时文件。

2.17 其它

G.OTH.01 不要包含无效或永不执行的代码

【描述】
本条款所讲的无效代码指的是删除后对程序结果无影响的代码，永不执行的代码指的是逻辑永远上执行不到的代码。这两种代码要么是编程错误要么是多余的代码，会干扰对代码的理解，应将其删除。

G.OTH.02 不要在信号处理函数中访问共享对象

【描述】
在信号处理程序中读写共享对象时，仅在访问无锁原子对象或volatile sig_atomic_t类型的对象时是安全的，读写其他共享对象可能会得到非预期结果。
此外，在信号处理程序中，如果要调用函数，仅可调用异步安全函数。

【反例】
在这个信号处理函数中，记录程序接收到SIGUSR1信号的次数，但是在信号处理函数中累加g_sigCount存在竞争条件，得到的结果可能不正确。

```
static long g_sigCount = 0;

static void Handler(int signum)
{
    // 下面代码操作存在竞争条件
    ++g_sigCount;
}

int main(void)
{
    // 设置SIGUSR1信号对应的处理函数
    if (signal(SIGUSR1, Handler) == SIG_ERR) {
        ... // 错误处理
    }
    ... // 程序主循环代码

    // 打印接收到信号的次数
    printf("%ld", g_sigCount);
```

```
    return 0;
}
```

【正例】
如下代码示例中，在信号处理函数中仅将`volatile sig_atomic_t`类型作为共享对象使用。

```
volatile sig_atomic_t g_sigCount = 0;

static void Handler(int signum)
{
    // 下面的操作是确定的
    ++g_sigCount;
}

int main(void)
{
    // 设置SIGUSR1信号对应的处理函数
    if (signal(SIGUSR1, Handler) == SIG_ERR) {
        ... // 错误处理
    }
    ... // 程序主循环代码

    // 打印接收到信号的次数
    printf("%ld", (long)g_sigCount);

    return 0;
}
```

G.OTH.03 禁用rand函数产生用于安全用途的伪随机数

【描述】
rand()函数生成的随机数是可以预测的，所以禁止使用rand()函数生成的随机数用于安全用途，必须使用安全的随机数生成方式，如：类Unix平台的/dev/random文件。

典型的安全用途场景包括（但不限于）以下几种：

- 会话标识SessionID的生成；
- 挑战算法中的随机数生成；
- 验证码的随机数生成；
- 用于密码算法用途（例如用于生成IV、盐值、密钥等）的随机数生成。

G.OTH.04 禁止在发布版本中输出对象或函数的地址

【描述】
禁止在发布版本中输出对象或函数的地址，如：将变量或函数的地址输出到客户端、日志、串口中。

当攻击者实施高级攻击时，通常需要先获取目标程序中的内存地址（如变量地址、函数地址等），再通过修改指定内存的内容，达到攻击目的。
如果程序中主动输出对象或函数的地址，则为攻击者提供了便利条件，可以根据这些地址以及偏移量计算出其他对象或函数的地址，并实施攻击。
另外，由于内存地址泄露，也会造成地址空间随机化的保护功能失效。

为降低地址泄露造成漏洞的风险，高版本linux内核代码中的printk函数已经将%p格式由打印地址数值修改为打印地址的hash值。
同时也提供了%pK %px等格式，应对特殊场景，因此内核中打印指针的策略可参考业界CVE修改漏洞的方法。

【反例】
如下代码中，使用%p格式将指针指向的地址记录到日志中。

```
int Encode(unsigned char *in, size_t inSize, unsigned char *out, size_t maxSize)
{
    ...
    Log("in=%p, in size=%zu, out=%p, max size=%zu\n", in, inSize, out, maxSize);
    ...
}
```

备注：这里仅用%p打印指针作为示例，代码中将指针转换为整数再打印也存在同样的风险。

【正例】
如下代码中，删除打印地址的代码。

```
int Encode(unsigned char *in, size_t inSize, unsigned char *out, size_t maxSize)
{
    ...
    Log("in size=%zu, max size=%zu\n", inSize, maxSize);
    ...
}
```

【例外】
当程序崩溃退出时，在记录崩溃的异常信息中可以输出内存地址等信息。

附录A 部分名词解释

名词	解释
API(Application Programming Interface)	应用软件编程接口：是指一些预先定义的函数，目的是提供应用程序与开发人员基于某软件或硬件得以访问一组例程的能力，而又无需访问源码或理解内部工作机制的细节。
DoS(Denial of Service)	拒绝服务：一种常用来使服务器或网络瘫痪的网络攻击手段。拒绝服务攻击者并不寻找进入内部网络的入口，而是阻止合法用户访问资源或设备。
RPC(Remote Procedure Call)	远程过程调用：是一个计算机通信协议。该协议允许运行于一台计算机的程序调用另一台计算机的子程序，而程序员无需额外地为这个交互作用编程。
污染数据(tainted data)	来自程序外部的输入。程序无法控制输入值。污染数据的来源可能是异常的用户输入，以及来自网络套接字或文件的数据流。污染数据可能会在应用程序中传播，并最终出现在多个不同的代码路径中。
悬空指针(dangling pointer)	指向曾经存在的对象的指针，但该对象已经不再存在了。
副作用(side effect)	函数或表达式的副作用是指除了函数返回值或表达式的运算结果外，还产生了其他可见的状态改变，如变量改变，磁盘写入，用户界面的按钮变化等。