文件编号：SYD/QP-RD-16

杭州信雅达科技有限公司

C 语言编码规范

（V1.00 版）

目录

1. 前言

为确保系统源程序可读性，提高产品代码质量，指导广大软件开发人员编写出简洁、可维护、可靠、可测试、高效、可移植的代码，编程规范修订工作组分析、总结了我司的各种典型编码问题，并参考了业界编程规范近年来的成果，对 C 语言编程规范进行了梳理、优化，编写了本规范。 系统继承的其它资源中的源程序也应按此规范作相应修改。

2. 使用范围

本文档将作为上门营销系统软件（C 语言）开发的编程格式规范。在系统的编码、测试及维护过程中，要求严格遵守。

3. 编程规范

3.1 头文件

3.1.1 .H 文件

每一个.c 文件应有一个同名.h 文件，用于声明需要对外公开的接口。 如果一个.c 文件不需要对外公布任何接口，则其就不应当存在，除非它是程序的入口，如 main 函数所在的文件。

现有某些产品中，习惯一个.c 文件对应两个头文件，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。编者不提倡这种风格。这种风格的根源在于源文件过大，应首先考虑拆分.c 文件，使之不至于太大。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 include 之，难以保证这些定义最后真的只是私有的。

本规则反过来并不一定成立。有些特别简单的头文件，如命令 ID 定义头文件，不需要有对应的.c 存在。 示例：对于如下场景，如在一个.c 中存在函数调用关系：

```
void foo()
{
   bar(); }
void bar() {
Do something; }
```

必须在 foo 之前声明 bar，否则会导致编译错误。

这一类的函数声明，应当在.c 的头部声明，并声明为 static 的，如下：

```
static void bar();
void foo() {
   bar();
}
void bar() {
Do something; }
```

3.1.2 禁止头文件循环依赖

头文件循环依赖，指 a.h 包含 b.h，b.h 包含 c.h，c.h 包含 a.h 之类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。

3.1.3 禁止包含用不到的头文件

.c/.h 文件禁止包含用不到的头文件，很多系统中头文件包含关系复杂，开发人员为了省事起见，可能不会去一一钻研，直接包含一切想到的头文件，甚至有些产品干脆发布了一个 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，导致整个系统的编译时间进一步恶化，并对后来人的维护造成了巨大的麻烦

3.1.4 头文件应当自包含

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

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“.c/.h 文件禁止包含用不到的头文件”规则一起使用，不能为了让 a.h 自包含，而在 a.h 中包含不必要的头文件。a.h 要刚刚可以自包含，不能在 a.h 中多包含任何满足自包含之外的其他头文件。

3.1.5 内部#INCLUDE 保护符

总是编写内部#include 保护符（#define 保护），多次包含一个头文件可以通过认真的设计来避免。如果不能做到这一点，就需要采取阻止头文件内容被包含多于一次的机制。

通常的手段是为每个文件配置一个宏，当头文件第一次被包含时就定义这个宏，并在头文件被再次包含时使用它以排除文件内容。

所有头文件都应当使用#define 防止头文件被多重包含，命名格式为 FILENAME_H，为了保证唯一性，更好的命名是 PROJECTNAME_PATH_FILENAME_H。

注：没有在宏最前面加上“_"，即使用 FILENAME_H 代替_FILENAME_H_，是因为一般以"_"和”__"开头的标识符为系统保留或者标准库使用，在有些静态检查工具中，若全局可见的标识符以"_"开头会给出告警。

定义包含保护符时，应该遵守如下规则：

1）保护符使用唯一名称；

2）不要在受保护部分的前后放置代码或者注释

示例：假定 VOS 工程的 timer 模块的 timer.h，其目录为 VOS/include/timer/timer.h,应按如下方式保护：

```
#ifndef VOS_INCLUDE_TIMER_TIMER_H
#define VOS_INCLUDE_TIMER_TIMER_H
  ...
#endif
```

也可以使用如下简单方式保护:

```
#ifndef TIMER_H
#define TIMER_H
  ..
#endif
```

例外情况：头文件的版权声明部分以及头文件的整体注释部分（如阐述此头文件的开发背景、使用注意事项等）可以放在保护符(#ifndef XX_H)前面。

3.1.6 禁止在头文件中定义变量

说明：在头文件中定义变量，将会由于头文件被其他.c 文件包含而导致变量重复定义。

3.1.7 禁止通过 EXTERN 使用外部函数

只能通过包含头文件的方式使用其他.c 提供的接口，禁止在.c 中通过 extern 的方式使用外部函数接口、变量。说明：若 a.c 使用了 b.c 定义的 foo()函数，则应当在 b.h 中声明 extern int foo(int input)；并在 a.c 中通过#include <b.h>来使用 foo。禁止通过在 a.c 中直接写 extern int foo(int input);来使用 foo，后面这种写法容易在 foo 改变时可能导致声明和定义不一致。

3.1.8 禁止在 EXTERN "C"中包含头文件

在 extern "C"中包含头文件，会导致 extern "C"嵌套，Visual Studio 对 extern "C"嵌套层次有限制，嵌套层次太多会编译错误。

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。例如，存在 a.h 和 b.h 两个头文件：

```
#ifndef A_H__
#define A_H__


#ifdef __cplusplus void foo(int);
#define a(value) foo(value)
#else
void a(int)
 #endif


#endif /* A_H__ */
```

```
#ifndef B_H__
#define B_H__


#ifdef __cplusplus  extern "C" {
#endif
#include "a.h"
      void b();
#ifdef __cplusplus
 }
#endif


#endif /* B_H__ */
```

使用 C++预处理器展开 b.h，将会得到

```
extern "C" {
```

```
    void foo(int);
    void b();
     }
```

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

示例：错误的使用方式：

```
extern “C”
 {
#include “xxx.h”
...
}
```

正确的使用方式：

```
#include “xxx.h”
 extern “C”
{
 ...
}
```

3.2 函数

3.2.1 避免函数过长

避免函数过长，新增函数不超过 50 行（非空非注释行），本规则仅对新增函数做要求，对已有函数修改时，建议不增加代码行。

过长的函数往往意味着函数功能不单一，过于复杂（参见原则 2.1：一个函数只完成一个功能）。函数的有效代码行数，即 NBNC（非空非注释行）应当在[1，50]区间。

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

延伸阅读材料：业界普遍认为一个函数的代码行不要超过一个屏幕，避免来回翻页影响阅读；一般的

代码度量工具建议都对此进行检查，例如 Logiscope 的函数度量："Number of Statement" （函数中

的可执行语句数）建议不超过 20 行，QA C 建议一个函数中的所有行数（包括注释和空白行）不超过 50 行。

3.2.2 避免嵌套过深

避免函数的代码块嵌套过深，新增函数的代码块嵌套不超过 4 层，本规则仅对新增函数做要求，对已有的代码建议不增加嵌套层次。

函数的代码块嵌套深度指的是函数中的代码控制块（例如：if、for、while、switch 等）之间互相包含的深度。每级嵌套都会增加阅读代码时的脑力消耗，因为需要在脑子里维护一个“栈”（比如，进入条件语句、进入循环„„）。应该做进一步的功能分解，从而避免使代码的阅读者一次记住太多的上下文。优秀代码参考值：[1, 4]。 示例：如下代码嵌套深度为 5。

```
void serial (void)
{
    if (!Received)
    {
        TmoCount = 0;
        switch (Buff)
        {
          case AISGFLG:
          if ((TiBuff.Count > 3)
                && ((TiBuff.Buff[0] == 0xff) || (TiBuf.Buff[0] == CurPa.ADDR)))
              {
                Flg7E = false;
                Received = true;
                }
                else
                {
                 TiBuff.Count = 0;
                 Flg7D = false;
                Flg7E = true;
```

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

3.2.3 避免使用共享变量

可重入函数应避免使用共享变量；若需要使用，则应通过互斥手段（关中断、信号量）对其加以保护。 可重入函数是指可能被多个任务并发调用的函数。在多任务操作系统中，函数具有可重入性是多个任务可以共用此函数的必要条件。共享变量指的全局变量和 static 变量。

编写 C 语言的可重入函数时，不应使用 static 局部变量，否则必须经过特殊处理，才能使函数具有可重入性。

示例：函数 square_exam 返回 g_exam 平方值。那么如下函数不具有可重入性。

```
int g_exam;
unsigned int example( int para )
{
   unsigned int temp;
   g_exam = para; // （**）
    temp = square_exam ( );
   return temp;
}
```

此函数若被多个线程调用的话，其结果可能是未知的，因为当（**）语句刚执行完后，另外一个使用本函数的线程可能正好被激活，那么当新激活的线程执行到此函数时，将使 g_exam 赋于另一个不同的 para 值，所以当控制重新回到“temp =square_exam ( )”后，计算出的 temp 很可能不是预想中的结果。此函数应如下改进。

```
int g_exam;
unsigned int example( int para )
```

```
{
   unsigned int temp;
   [申请信号量操作]  // 若申请不到“信号量”，说明另外的进程正处于
   g_exam = para;     //给 g_exam 赋值并计算其平方过程中（即正在使用此
   temp = square_exam( ); // 信号），本进程必须等待其释放信号后，才可继
   [释放信号量操作]  // 续执行。其它线程必须等待本线程释放信号量后
   // 才能再使用本信号。
   return temp;
}
```

3.2.4 参数的合法性检查

对参数的合法性检查，由调用者负责还是由接口函数负责，应在项目组/模块内应统一规定。缺省由调用者负责。

说明：对于模块间接口函数的参数的合法性检查这一问题，往往有两个极端现象，即：要么是调用者和被调用者对参数均不作合法性检查，结果就遗漏了合法性检查这一必要的处理过程，造成问题隐患；要么就是调用者和被调用者均对参数进行合法性检查，这种情况虽不会造成问题，但产生了冗余代码，降低了效率。

示例：下面红色部分的代码在每一个函数中都写了一次，导致代码有较多的冗余。如果函数的参数比较多，而且判断的条件比较复杂（比如：一个整形数字需要判断范围等），那么冗余的代码会大面积充斥着业务代码。

```
void PidMsgProc(MsgBlock *Msg)
{
   MsgProcItem *func = NULL
  if (Msg == NULL)
{
      return;
}
   ... ...
   GetMsgProcFun(Msg, &func);
```

```
    func(Msg);
    return;
}
int GetMsgProcFun(MsgBlock *Msg, MsgProcItem **func)
{
    if (Msg == NULL)
    {
        return 1;
    }
    ... ...
    *func = VOS_NULL_PTR;
    for (Index = 0; Index < NELEM(g_MsgProcTable); Index++)
{
        if ((g_MsgProcTable[Index].FlowType == Msg->FlowType)
                && (g_MsgProcTable[Index].Status == Msg->Status)
                && (g_MsgProcTable[Index].MsgType == Msg->MsgType))
                {
            *func = &(g_MsgProcTable[Index]);
                return 0;
                    }
        }
    return 1;
}
int ServiceProcess(int CbNo, MsgBlock *Msg) {
    if ( Msg == NULL)    {
        return 1;    }
    ... ...
    // 业务处理代码
     ... ...
```

```
    return 0;
}
```

3.2.5 错误返回码

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

```
FILE *fp = fopen( "./writeAlarmLastTime.log","r");
char buff[128] = "";
fscanf(fp,“%s”, buff); /* 读取最新的告警时间；由于文件 writeAlarmLastTime.log 为空，导致 buff 为空 */ fclose(fp);
long fileTime = getAlarmTime(buff); /* 解析获取最新的告警时间；getAlarmTime 函数未检查 buff 指针，导致宕机 */
```

正确写法：

```
FILE *fp = fopen( "./writeAlarmLastTime.log","r");
char buff[128] = "";
if (fscanf(fp,“%s”,buff) == EOF) //检查函数 fscanf 的返回值，确保读到数据
{
return
}
fclose(fp);
long fileTime = getAlarmTime(buff); //解析获取最新的告警时间；
```

3.2.6 废弃代码清除

废弃代码（没有被调用的函数和变量)要及时清除。，程序中的废弃代码不仅占用额外的空间，而

且还常常影响程序的功能与性能，很可能给程序的测试、维护等造成不必要的麻烦。

3.3 标示符命名与定义

3.3.1 通用命名规则

目前比较使用的如下几种命名风格：

unix like 风格：此风格适用于 单词用小写字母，每个单词直接用下划线“_”分割，例如 syd_text_mutex，kernel_text_address。

函数示例： ReadFileInfo()

变量示例： SYD_SM2_EncData

Windows 风格：大小写字母混用，单词连在一起，每个单词首字母大写。不过 Windows 风格如果遇到大写专有用语时会有些别扭，例如命名一个读取 RFC 文本的函数，命令为 ReadRFCText，看起来就没有 unix like 的 read_rfc_text 清晰了。

标识符的命名要清晰、明了，有明确含义，同时使用完整的单词或大家基本可以理解的缩写，避免使人产生误解。，尽可能给出描述性名称，不要节约空间，让别人很快理解你的代码更重要。

示例：好的命名：

```
int error_number;
int number_of_completed_connection;
```

不好的命名：使用模糊的缩写或随意的字符：

```
int n;
int nerr;
int n_comp_conns;
```

产品/项目组内部应保持统一的命名风格，Unix like 和 windows like 风格均有其拥趸，产品应根据自己的部署平台，选择其中一种，并在产品内部保持一致。 例外：即使产品之前使用匈牙利命名法，新代码也不应当使用。

3.3.2 文件命名规则

文件命名统一采用小写字符，因为不同系统对文件名大小写处理会不同（如 MS 的 DOS、Windows 系统不区分大小写，但是 Linux 系统则区分），所以代码文件命名建议统一采用全小写字母命名。

3.3.3 变量命名规则

全局变量应增加“g_”前缀，静态变量应增加“s_”前缀。

原因如下：

首先，全局变量十分危险，通过前缀使得全局变量更加醒目，促使开发人员对这些变量的使用更加小心。

其次，从根本上说，应当尽量不使用全局变量，增加 g_和 s_前缀，会使得全局变量的名字显得很丑陋，从而促使开发人员尽量少使用全局变量。

禁止使用单字节命名变量，但运行定义 i、j、k 作为局部循环变量。

3.3.4 函数命名规则

函数命名应以函数要执行的动作命名，一般采用动词或者动词＋名词的结构。

示例：找到当前进程的当前目录

DWORD GetCurrentDirectory(DWORD BufferLength, LPTSTR Buffer);

函数指针除了前缀，其他按照函数的命名规则命名。

3.3.5 宏的命名规则

对于数值或者字符串等等常量的定义，建议采用全大写字母，单词之间加下划线„_“的方式命名（枚举同样建议使用此方式定义）。 示例：

#define PI_ROUNDED 3.14

除了头文件或编译开关等特殊标识定义，宏定义不能使用下划线„_“开头和结尾。 一般来说，“_”开头、结尾的宏都是一些内部的定义， ISO/IEC 9899（俗称 C99）中有如下的描述（6.10.8 Predefined macro names）：

None of these macro names （这里上面是一些内部定义的宏的描述）, nor the identifier defined, shall be the subject of a #define or a #undef preprocessing directive. Any other predefined macro names shall begin with a leading underscore followed by an uppercase letter or a second underscore.

3.4 变量

一个变量只有一个功能，不能把一个变量用作多种用途。

说明：一个变量只用来表示一个特定功能，不能把一个变量作多种用途，即同一变量取值不同时，其代表的意义也不同。

示例：具有两种功能的反例

```
WORD DelRelTimeQue(void)
{
    WORD Locate;
    Locate = 3;
    Locate = DeleteFromQue(Locate); /* Locate 具有两种功能：位置和函数 DeleteFromQue 的返回值 */
    return Locate;
}
```

正确做法：使用两个变量

```
WORD  DelRelTimeQue(void)
{
    WORD Ret;  WORD Locate;
    Locate  =  3;
    Ret   =   DeleteFromQue(Locate);
     return wValue;
}
```

3.4.1 单一的数据结构

结构功能单一；不要设计面面俱到的数据结构。相关的一组信息才是构成一个结构体的基础，结构的定义应该可以明确的描述一个对象，而不是一组相关性不强的数据的集合。

设计结构时应力争使结构代表一种现实事务的抽象，而不是同时代表多种。结构中的各元素应代表同一事务的不同侧面，而不应把描述没有关系或关系很弱的不同事务的元素放到同一结构中。

示例：如下结构不太清晰、合理。

```
typedef struct STUDENT_STRU
{
unsigned char name[32];   /* student's name */
unsigned char age;            /* student's age */
unsigned char sex;            /* student's sex, as follows */
   /* 0 - FEMALE; 1 - MALE */
 unsigned char teacher_name[32]; /* the student teacher's name */
 unsigned char teacher_sex;     /* his teacher sex */
} STUDENT;
```

若改为如下，会更合理些。

```
typedef struct TEACHER_STRU
 {
unsigned char name[32];     /* teacher name */
 unsigned char sex;              /* teacher sex, as follows */
/* 0 - FEMALE; 1 - MALE */
 unsigned int teacher_ind;   /* teacher index */
} TEACHER;

typedef struct STUDENT_STRU
{
unsigned char name[32];     /* student's name */
unsigned char age;              /* student's age */
 unsigned char sex;              /* student's sex, as follows */
/* 0 - FEMALE; 1 - MALE */
 unsigned int  teacher_ind;   /* his teacher index */
} STUDENT;
```

3.4.2 不用或者少用全局变量。

单个文件内部可以使用 static 的全局变量，可以将其理解为类的私有成员变量。

全局变量应该是模块的私有数据，不能作用对外的接口使用，使用 static 类型定义，可以有效防止外部文件的非正常访问，建议定义一个 STATIC 宏，在调试阶段，将 STATIC 定义为 static，版本发布时，改为空，以便于后续的打补丁等操作。

```
#ifdef _DEBUG
#define STATIC static #else
#define STATIC #endif
```

直接使用其他模块的私有数据，将使模块间的关系逐渐走向“剪不断理还乱”的耦合状态，这种情形是不允许的。

3.5 宏、常量

3.5.1 宏定义表达式

建议降低宏定义表达式处理的歧义性。

用宏定义表达式时，要使用完备的括号。因为宏只是简单的代码替换，不会像函数一样先将参数计算后，再传递。

示例：如下定义的宏都存在一定的风险

```
#define RECTANGLE_AREA(a, b) a * b
#define RECTANGLE_AREA(a, b) (a * b)
#define RECTANGLE_AREA(a, b) (a) * (b)
```

正确的定义应为：

```
#define RECTANGLE_AREA(a, b) ((a) * (b))
```

这是因为：

如果定义

#define RECTANGLE_AREA(a, b) a * b 或#define RECTANGLE_AREA(a, b) (a * b)

则 c/RECTANGLE_AREA(a, b) 将扩展成 c/a * b , c 与 b 本应该是除法运算，结果变成了乘法运算，造成错误。

如果定义#define RECTANGLE_AREA(a, b) (a) * (b)

则 RECTANGLE_AREA(c + d, e + f)将扩展成：(c + d * e + f), d 与 e 先运算，造成错误。

3.5.2 宏定义的多条表达式

将宏所定义的多条表达式放在大括号中，更好的方法是多条语句写成 do while(0)的方式。

示例：看下面的语句，只有宏的第一条表达式被执行。

```
 #define FOO(x) \
printf("arg is %d\n", x); \
do_something_useful(x);
```

为了说明问题，下面 for 语句的书写稍不符规范

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

用大括号定义的方式可以解决上面的问题：

```
#define FOO(x) { \
      printf("arg is %s\n", x); \
       do_something_useful(x); \
}
```

但是如果有人这样调用：

```
 if (condition == 1)
      FOO(10);
else
      FOO(20);
```

那么这个宏还是不能正常使用，所以必须这样定义才能避免各种问题：

```
#define FOO(x)   do   { \
printf("arg is %s\n", x);   \
do_something_useful(x);   \
} while(0)
```

用 do-while(0)方式定义宏，完全不用担心使用者如何使用宏，也不用给使用者加什么约束。

3.5.3 禁止参数变化

使用宏时，不允许参数发生变化。

示例：如下用法可能导致错误。

```
#define SQUARE(a) ((a) * (a))
int a = 5;
int b;
b = SQUARE(a++); // 结果：a = 7，即执行了两次增。
```

正确的用法是：

```
b = SQUARE(a);
a++; // 结果：a = 6，即只执行了一次增。
```

同时也建议即使函数调用，也不要在参数中做变量变化操作，因为可能引用的接口函数，在某个版本升级后，变成了一个兼容老版本所做的一个宏，结果可能不可预知。

3.5.4 禁止直接使用魔鬼数字

使用魔鬼数字的弊端：代码难以理解；如果一个有含义的数字多处使用，一旦需要修改这个数值，代价惨重。

使用明确的物理状态或物理意义的名称能增加信息，并能提供单一的维护点。

解决途径：

对于局部使用的唯一含义的魔鬼数字，可以在代码周围增加说明注释，也可以定义局部 const 变量，变量命名自注释。

对于广泛使用的数字，必须定义 const 全局变量/宏；同样变量/宏命名应是自注释的。 0 作为一个特殊的数字，作为一般默认值使用没有歧义时，不用特别定义。

3.6 质量保证

3.6.1 禁止内存操作越界。

说明：内存操作主要是指对数组、指针、内存地址等的操作。内存操作越界是软件系统主要错误之一，后果往往非常严重，所以当我们进行这些操作时一定要仔细小心。

示例：使用 itoa（）将整型数转换为字符串时：

```
char TempShold[10]
itoa(ProcFrecy,TempShold, 10); /* 数据库刷新间隔设为值 1073741823 时，系统监控后台 coredump, 监控前台抛异常。*/
```

TempShold 是以‘\0’结尾的字符数组，只能存储 9 个字符，而 ProcFrecy 的最大值可达到 10 位，导致符数组 TempShold 越界。

正确写法：一个 int（32 位）在－2147483647～2147483648 之间，将数组 TempShold 设置成 12 位。

```
char TempShold[12]
itoa(ProcFrecy,TempShold,10);
```

坚持下列措施可以避免内存越界：

- 数组的大小要考虑最大情况，避免数组分配空间不够。
- 避免使用危险函数 sprintf /vsprintf/strcpy/strcat/gets 操作字符串，使用相对安全的函数 snprintf/strncpy/strncat/fgets 代替。
- 使用 memcpy/memset 时一定要确保长度不要越界
- 字符串考虑最后的’\0’，确保所有字符串是以’\0’结束
- 指针加减操作时，考虑指针类型长度
- 数组下标进行检查
- 使用时 sizeof 或者 strlen 计算结构/字符串长度，避免手工计算

3.6.2 禁止内存泄漏

内存和资源（包括定时器/文件句柄/Socket/队列/信号量/GUI 等各种资源）泄漏是常见的错误。示例：异常出口处没有释放内存

```
MsgDBDEV = (PDBDevMsg)GetBuff( sizeof( DBDevMsg ), __LINE__);
if (MsgDBDEV == NULL)
{
   return;
}
MsgDBAppToLogic = (LPDBSelfMsg)GetBuff( sizeof(DBSelfMsg), __LINE__ );
```

```
if ( MsgDBAppToLogic == NULL )
{
   return; //MsgDB_DEV 指向的内存丢失
}
```

坚持下列措施可以避免内存泄漏：

- 异常出口处检查内存、定时器/文件句柄/Socket/队列/信号量/GUI 等资源是否全部释放
- 删除结构指针时，必须从底层向上层顺序删除
- 使用指针数组时，确保在释放数组时，数组中的每个元素指针是否已经提前被释放了
- 避免重复分配内存
- 小心使用有 return、break 语句的宏，确保前面资源已经释放
- 检查队列中每个成员是否释放

3.6.3 禁止引用释放的内存

在实际编程过程中，稍不留心就会出现在一个模块中释放了某个内存块，而另一模块在随后的某个时刻又使用了它。要防止这种情况发生。

示例：一个函数返回的局部自动存储对象的地址，导致引用已经释放的内存空间

```
int* foobar (void)
{
int local_auto = 100;
return &local_auto;
}
```

坚持下列措施可以避免引用已经释放的内存空间：

- 内存释放后，把指针置为 NULL；使用内存指针前进行非空判断。
- 耦合度较强的模块互相调用时，一定要仔细考虑其调用关系，防止已经删除的对象被再次使用。
- 避免操作已发送消息的内存。
- 自动存储对象的地址不应赋值给其他的在第一个对象已经停止存在后仍然保持的对象（具有更大作用域的对象或者静态对象或者从一个函数返回的对象）

3.6.4 防止差 1 错误

编程时，防止差 1 错误一般是由于把“<=”误写成“<”或“>=”误写成“>”等造成的，由此引起的后果，很多情况下是很严重的，所以编程时，一定要在这些地方小心。当编完程序后，应对这些操作符进行彻底检查。使用变量时要注意其边界值的情况。

示例：如 C 语言中字符型变量，有效值范围为-128 到 127。故以下表达式的计算存在一定风险。

```
char ch = 127;
int sum = 200;
ch += 1; // 127 为 ch 的边界值，再加将使 ch 上溢到-128，而不是 128
sum += ch; // 故 sum 的结果不是 328，而是 72。
```

3.6.5 禁止滥用 GOTO 语句

goto 语句会破坏程序的结构性，所以除非确实需要，最好不使用 goto 语句。

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

```
int foo(void)
{
   char* p1 = NULL;
   char* p2 = NULL;
   char* p3 = NULL;
   int   result = -1;
   p1 = (char *)malloc(0x100);
   if (p1 == NULL)
   {
      goto Exit0;
```

```
}
strcpy(p1, "this is p1");
p2 = (char *)malloc(0x100);
if (p2 == NULL)
{
   goto Exit0;
}
strcpy(p2, "this is p2");
p3 = (char *)malloc(0x100);
if (p3 == NULL)
{
   goto Exit0;
}
strcpy(p3, "this is p3");
result = 0;
Exit0:
free(p1); // C 标准规定可以 free 空指针
free(p2);
free(p3);
return result;
}
```

3.7 程序效率

在保证软件系统的正确性、简洁、可维护性、可靠性及可测性的前提下，提高代码效率。 本章节后面所有的规则和建议，都应在不影响前述可读性等质量属性的前提下实施。

不能一味地追求代码效率，而对软件的正确、简洁、可维护性、可靠性及可测性造成影响。 产品代码中经常有如下代码：

```
int foo()
{
   if (异常条件)
{
      异常处理;
      return ERR_CODE_1;
}
   if (异常条件）
{
      异常处理;
      return ERR_CODE_2;
}
   正常处理;
   return SUCCESS;
}
```

这样的代码看起来很清晰，而且也避免了大量的 if else 嵌套。但是从性能的角度来看，应该把执行概率较大的分支放在前面处理，由于正常情况下的执行概率更大，若首先考虑性能，应如下书写：

```
int foo()
{
   if (满足条件)
   {
         正常处理;
         return SUCCESS;
```

```
        }
          else if (概率比较大的异常条件)
        {
             异常处理;
             return ERR_CODE_1;
        }
        else
        {
             异常处理;
             return ERR_CODE_2;
         }
}
```

除非证明 foo 函数是性能瓶颈，否则按照本规则，应优先选用前面一种写法。

以性能为名，使设计或代码更加复杂，从而导致可读性更差，但是并没有经过验证的性能要求（比如实际的度量数据和目标的比较结果）作为正当理由，本质上对程序没有真正的好处。无法度量的优化行为其实根本不能使程序运行得更快。

记住：让一个正确的程序更快速，比让一个足够快的程序正确，要容易得太多。大多数时候，不要把注意力集中在如何使代码更快上，应首先关注让代码尽可能地清晰易读和更可靠。

3.8 注释

3.8.1 头部文件注释

文件头部应进行注释，注释必须列出：版权说明、版本号、生成日期、作者姓名、内容、功能说明、与其它文件的关系、修改日志等，头文件的注释中还应有函数功能简要说明。通常头文件要对功能和用法作简单说明，源文件包含了更多的实现细节或算法讨论。 版权声明格式：Copyright © Huawei Technologies Co., Ltd. 1998-2011. All rights reserved. 1998-2011 根据实际需要可以修改， 1998 是文件首次创建年份，而 2011 是最新文件修改年份。

示例：下面这段头文件的头注释比较标准，当然，并不局限于此格式，但上述信息建议要包含在内。

```
/**************************************************
Copyright © Huawei Technologies Co., Ltd. 1998-2011. All rights reserved.
File name:      // 文件名
Author:             Version:          Date:       // 作者、版本及创建日期
 Description:  // 用于详细说明此程序文件完成的主要功能，与其他模块
                    // 或函数的接口，输出值、取值范围、含义及参数间的控
                    // 制、顺序、独立或依赖等关系
Others:          // 其它内容的说明
History:         // 修改历史记录列表，每条修改记录应包括修改日期、修改
                    // 作者及修改内容简述
    1. Date:
       Author:
       Modification:
    2. ...
**************************************************/
```

3.8.2 函数声明注释

函数声明处注释描述函数功能，包括功能说明、输入和输出参数、函数返回值等，如果有其它重点约束需要说明的，可在备注中说明。

备注：其它重要提示。

3.8.3 全局变量注释

全局变量应尽量不用或少用，若需要使用，则全局变量第一个字母必须为小写的 g_变量名。

全局变量要有较详细的注释，包括对其功能、取值范围以及存取时注意事项等的说明。

示例：

```
/* The ErrorCode when SCCP translate */
/* Global Title failure, as follows */     /* 变量作用、含义*/
```

```
/* 0 －SUCCESS   1 －GT Table error */
/* 2 －GT error  Others －no use  */      /* 变量取值范围*/
/* only  function  SCCPTranslate() in */
/* this modual can modify it,  and  other */
/* module can visit it through call */
/* the  function GetGTTransErrorCode() */    /* 使用方法*/
BYTE g_GTTranErrorCode;
```

3.8.4 注释位置

注释应放在其代码上方相邻位置或右方，不可放在下面。如放于上方则需与其上面的代码用空行隔开，且与下方代码缩进相同。 示例：

```
/* active statistic task number */
#define MAX_ACT_TASK_NUMBER 1000
#define MAX_ACT_TASK_NUMBER 1000 /* active statistic task number */
可按如下形式说明枚举/数据/联合结构。
/* sccp interface with sccp user primitive message name */
enum SCCP_USER_PRIMITIVE
{
   N_UNITDATA_IND, /* sccp notify sccp user unit data come */
   N_NOTICE_IND, /* sccp notify user the No.7 network can not transmission this message */
   N_UNITDATA_REQ, /* sccp user's unit data transmission request*/
};
```

3.8.5 CASE 语句的注释

对于 switch 语句下的 case 语句，如果因为特殊情况需要处理完一个 case 后进入下一个 case 处理，必须在该 case 语句处理完、下一个 case 语句前加上明确的注释。 说明：这样比较清楚程序编写者的意图，有效防止无故遗漏 break 语句。

示例（注意斜体加粗部分）：

```
case CMD_FWD:
ProcessFwd();
   /* now jump into case CMD_A */
case CMD_A:
      ProcessA();
      break;
      //对于中间无处理的连续 case，已能较清晰说明意图，不强制注释。
switch (cmd_flag)
{
case CMD_A:
case CMD_B:
      {
            ProcessCMD();
            break;
      }
      ……
}
```

3.9 排版与格式

3.9.1 缩进风格

程序块采用缩进风格编写，每级缩进为 4 个空格。

说明：当前各种编辑器/IDE 都支持 TAB 键自动转空格输入，需要打开相关功能并设置相关功能。编辑器/IDE 如果有显示 TAB 的功能也应该打开，方便及时纠正输入错误。 IDE 向导生成的代码可以不用修改。

宏定义、编译开关、条件预处理语句可以顶格（或使用自定义的排版方案，但产品/模块内必须保持一致）。

3.9.2 空行风格

相对独立的程序块之间、变量说明之后必须加空行。

示例：如下例子不符合规范。

```
if (!valid_ni(ni))
{
   // program code
      ...
}
repssn_ind = ssn_data[index].repssn_index;
repssn_ni = ssn_data[index].ni;
```

应如下书写

```
if (!valid_ni(ni))
{
   // program code
      ...
}

repssn_ind = ssn_data[index].repssn_index;
repssn_ni = ssn_data[index].ni;
```

3.9.3 换行风格

一条语句不能过长，如不能拆分需要分行写。一行到底多少字符换行比较合适，产品可以自行确定。 对于目前大多数的 PC 来说，132 比较合适（80/132 是 VTY 常见的行宽值）；对于新 PC 宽屏显示器较多的产品来说，可以设置更大的值。 换行时有如下建议：

- 换行时要增加一级缩进，使代码可读性更好；
- 低优先级操作符处划分新行；换行时操作符应该也放下来，放在新行首；
- 换行时建议一个完整的语句放在一行，不要根据字符数断行

示例：

```
if ((temp_flag_var == TEST_FLAG)
   &&(((temp_counter_var - TEST_COUNT_BEGIN) % TEST_COUNT_MODULE) >= TEST_COUNT_THRESH
OLD))
{
   // process code
}
```

3.9.4 多语句风格

多个短语句（包括赋值语句）不允许写在同一行内，即一行只写一条语句。

示例：

```
int a = 5; int b= 10;  //不好的排版
```

较好的排版

```
int a = 5
int b= 10;
```

3.9.5 循环语句风格

if、for、do、while、case、switch、default 等语句独占一行。执行语句必须用缩进风格写，属于 if、for、do、while、case、switch、default 等下一个缩进级别；

一般写 if、for、do、while 等语句都会有成对出现的„{}‟，对此有如下建议可以参考：

if、for、do、while 等语句后的执行语句建议增加成对的„{}‟；

如果 if/else 配套语句中有一个分支有„{}‟，那么令一个分支即使一行代码也建议增加„{}‟；

添加„{‟的位置可以在 if 等语句后，也可以独立占下一行；独立占下一行时，可以和 if 在一个缩进级别，也可以在下一个缩进级别；但是如果 if 语句很长，或者已经有换行，建议„{‟使用独占一行的写法。

3.9.6 空格风格

在两个以上的关键字、变量、常量进行对等操作时，它们之间的操作符之前、之后或者前后要加空格；进行非对等操作时，如果是关系密切的立即操作符（如一>），后不应加空格。 说明：采用这种松散方式编写代码的目的是使代码更加清晰。

在已经非常清晰的语句中没有必要再留空格，如括号内侧(即左括号后面和右括号前面)不需要加空格，多重括号间不必加空格，因为在 C 语言中括号已经是最清晰的标志了。

在长语句中，如果需要加的空格非常多，那么应该保持整体清晰，而在局部不加空格。给操作符留空格时不要连续留两个以上空格。

示例：

(1) 逗号、分号只在后面加空格

```
int a, b, c;
```

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

```
if (current_time >= MAX_TIME_VALUE)
a = b + c;
a *= 2;
a = b ^ 2;
```

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

```
*p = 'a';              // 内容操作"*"与内容之间
flag = !is_empty; // 非操作"!"与内容之间
p = &mem;          // 地址操作"&" 与内容之间
i++;                    // "++","--"与内容之间
```

(4) "->"、"."前后不加空格。

```
p->id = pid;          // "->"指针前后不加空格
```

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

```
if (a >= b && c > d)
```

3.10 表达式

表达式的值在标准所允许的任何运算次序下都应该是相同的。 除了少数操作符（函数调用操作符 ( )、&&、||、? : 和 ,（逗号）） 之外，子表达式所依据的运算次序是未指定的并会随时更改。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。

将复合表达式分开写成若干个简单表达式，明确表达式的运算次序，就可以有效消除非预期副作用。

1、自增或自减操作符

示例：

```
x = b[i] + i++;
```

b[i] 的运算是先于还是后于 i ++ 的运算，表达式会产生不同的结果，把自增运算做为单独的语句，可以避免这个问题。

```
x = b[i] + i;
i ++;
```

2、函数参数

说明：函数参数通常从右到左压栈，但函数参数的计算次序不一定与压栈次序相同。

示例：

```
x = func( i++, i);
```

应该修改代码明确先计算第一个参数： i++;

```
x = func(i, i);
```

3、函数指针

说明：函数参数和函数自身地址的计算次序未定义。 示例：

```
p->task_start_fn(p++);
```

求函数地址 p 与计算 p++无关，结果是任意值。必须单独计算 p++：

```
p->task_start_fn(p);
p++;
```

4、函数调用

示例：

```
int g_var = 0;
int fun1()
{
   g_var += 10;
   return g_var;
}
int fun2()
{
    g_var += 100;
   return g_var;
}
int x = fun1() + fun2();
```

编译器可能先计算 fun1()，也可能先计算 fun2()，由于 x 的结果依赖于函数 fun1()/fun2()的计算次序（fun1()/fun2()被调用时修改和使用了同一个全局变量），则上面的代码存在问题。 应该修改代码明确 fun1/ fun2 的计算次序：

```
int x = fun1();
x = x + fun2();
```

5、嵌套赋值语句

说明：表达式中嵌套的赋值可以产生附加的副作用。不给这种能导致对运算次序的依赖提供任何机会的最好做法是，不要在表达式中嵌套赋值语句。 示例：

```
x = y = y = z / 3;
x = y = y++;
```

6、volatile 访问

说明：限定符 volatile 表示可能被其它途径更改的变量，例如硬件自动更新的寄存器。编译器不会优化对 volatile 变量的读取。

示例：下面的写法可能无法实现作者预期的功能：

/* volume 变量被定义为 volatile 类型*/

UINT16 x = (volume << 3) | volume;

/* 在计算了其中一个子表达式的时候，volume 的值可能已经被其它程序或硬件改变，导致另外一个子表达式的计算结果非预期，可能无法实现作者预期的功能*/

3.10.1函数调用

函数调用不要作为另一个函数的参数使用，否则对于代码的调试、阅读都不利。如下代码不合理，仅用于说明当函数作为参数时，由于参数压栈次数不是代码可以控制的，可能造成未知的输出：

```
int g_var;  int fun1()
{
   g_var += 10;
   return g_var;
}
int fun2()
{
   g_var += 100;
   return g_var;
   }
int main(int argc, char *argv[], char *envp[])
{
   g_var = 1;
   printf("func1: %d, func2: %d\n", fun1(), fun2());
   g_var = 1;
   printf("func2: %d, func1: %d\n", fun2(), fun1());
}
```

上面的代码，使用断点调试起来也比较麻烦，阅读起来也不舒服，所以不要为了节约代码行，而写这种代码。

3.10.2赋值语句

赋值语句不要写在 if 等语句中，或者作为函数的参数使用。因为 if 语句中，会根据条件依次判断，如果前一个条件已经可以判定整个条件，则后续条件语句不会再运行，所以可能导致期望的部分赋值没有得到运行。

示例：

```
int main(int argc, char *argv[], char *envp[])
{
   int a = 0;     int b;
   if ((a == 0) || ((b = fun1()) > 10))
   {
      printf("a: %d\n", a);
   }
   printf("b: %d\n", b);
}
```

作用函数参数来使用，参数的压栈顺序不同可能导致结果未知。 看如下代码，能否一眼看出输出结果会是什么吗？好理解吗？

```
int g_var;
int main(int argc, char *argv[ ], char *envp[ ] )
{
   g_var = 1;
   printf("set 1st: %d, add 2nd: %d\n", g_var = 10, g_var++);
   g_var = 1;
   printf("add 1st: %d, set 2nd: %d\n", g_var++, g_var = 10);
}
```

3.10.3表达式顺序

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

1. 一元操作符，不需要使用括号

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

2. 二元以上操作符，如果涉及多种操作符，则应该使用括号

```
x = a + b + c;       /* 操作符相同，不需要括号*/
x = f ( a + b, c )   /* 操作符相同，不需要括号*/
if (a && b && c) /* 操作符相同，不需要括号*/
x = (a * 3) + c + d; /* 操作符不同，需要括号*/
x = ( a == b ) ? a : ( a －b );  /* 操作符不同，需要括号*/
```

3 .即使所有操作符都是相同的，如果涉及类型转换或者量级提升，也应该使用括号控制计算的次序 以下代码将 3 个浮点数相加：

/* 除了逗号(,)，逻辑与(&&)，逻辑或(||)之外，C 标准没有规定同级操作符是从左还是从右开始计算，以上表达式存在种计算次序：f4 = (f1 + f2) + f3 或 f4 = f1 + (f2 + f3)，浮点数计算过程中可能四舍五入，量级提升，计算次序的不同会导致 f4 的结果不同，以上表达式在不同编译器上的计算结果可能不一样，建议增加括号明确计算顺序*/

```
f4 =  f1 + f2 + f3;
```

3.11 代码编辑、编译

使用编译器的最高告警级别，理解所有的告警，通过修改代码而不是降低告警级别来消除所有告警。

在产品软件（项目组）中，要统一编译开关、静态检查选项以及相应告警清除策略。如果必须禁用某个告警，应尽可能单独局部禁用，并且编写一个清晰的注释，说明为什么屏蔽。 某些语句经编译/静态检查产生告警，但如果你认为它是正确的，那么应通过某种手段去掉告警信息。

本地构建工具（如 PC-Lint）的配置应该和持续集成的一致。两者一致，避免经过本地构建的代码在持续集成上构建失败。

使用版本控制（配置管理）系统，及时签入通过本地构建的代码，确保签入的代码不会影响构建成功，及时签入代码降低集成难度。

3.12 安全性

代码的安全漏洞大都是由代码缺陷导致，但不是所有代码缺陷都有安全风险。理解安全漏洞产生的原理和如何进行安全编码是减少软件安全问题最直接有效的办法。对用户输入进行检查。

不能假定用户输入都是合法的，因为难以保证不存在恶意用户，即使是合法用户也可能由于误用误操作而产生非法输入。用户输入通常需要经过检验以保证安全，特别是以下场景：

- 用户输入作为循环条件
- 用户输入作为数组下标
- 用户输入作为内存分配的尺寸参数
- 用户输入作为格式化字符串
- 用户输入作为业务数据（如作为命令执行参数、拼装 sql 语句、以特定格式持久化）

这些情况下如果不对用户数据做合法性验证，很可能导致 DOS、内存越界、格式化字符串漏洞、命令注入、SQL 注入、缓冲区溢出、数据破坏等问题。 可采取以下措施对用户输入检查：

- 用户输入作为数值的，做数值范围检查
- 用户输入是字符串的，检查字符串长度
- 用户输入作为格式化字符串的，检查关键字“%”
- 用户输入作为业务数据，对关键字进行检查、转义

3.12.1字符串操作安全

3.12.1.1字符串以 NULL 结束

C 语言中"\0"作为字符串的结束符，即 NULL 结束符。标准字符串处理函数（如 strcpy()、strlen()）

依赖 NULL 结束符来确定字符串的长度。没有正确使用 NULL 结束字符串会导致缓冲区溢出和其它未定义的行为。

为了避免缓冲区溢出，常常会用相对安全的限制字符数量的字符串操作函数代替一些危险函数。如：

- 用 strncpy()代替 strcpy()
- 用 strncat()代替 strcat()
- 用 snprintf()代替 sprintf()
- 用 fgets()代替 gets()

这些函数会截断超出指定限制的字符串，但是要注意它们并不能保证目标字符串总是以 NULL 结尾。如果源字符串的前 n 个字符中不存在 NULL 字符，目标字符串就不是以 NULL 结尾。

示例：

```
char a[16];
strncpy(a, "0123456789abcdef", sizeof(a));
```

上述代码存在安全风险：在调用 strncpy()后，字符数组 a 中的字符串是没有 NULL 结束符的，也没有空间存放 NULL 结束符。

正确写法：截断字符串，保证字符串以 NULL 结束。

```
char a[16];
strncpy(a, "0123456789abcdef", sizeof(a) - 1 );
a[sizeof(a) - 1] = '\0';
```

3.12.1.2字符串和数组

不要将边界不明确的字符串写到固定长度的数组中。边界不明确的字符串（如来自 gets()、getenv()、scanf()的字符串），长度可能大于目标数组长度，直接拷贝到固定长度的数组中容易导致缓冲区溢出。示例：

```
char buff[256];
char *editor = getenv("EDITOR");
if (editor != NULL)
{
   strcpy(buff, editor);
```

```
}
```

上述代码读取环境变量"EDITOR"的值，如果成功则拷贝到缓冲区 buff 中。而从环境变量获取到的字符串长度是不确定的，把它们拷贝到固定长度的数组中很可能导致缓冲区溢出。

正确写法：计算字符串的实际长度，使用 malloc 分配指定长度的内存

```
char *buff;
char *editor = getenv("EDITOR");
if (editor != NULL)
{
   buff = malloc(strlen(editor) + 1);
   if (buff != NULL)
   {
      strcpy(buff, editor);
   }
}
```

3.12.2整数安全

C99 标准定义了整型提升（integer promotions）、整型转换级别（integer conversion rank）以及普通算术转换（usual arithmetic conversions）的整型操作。不过这些操作实际上也带来了安全风险。

3.12.2.1避免整数溢出

当一个整数被增加超过其最大值时会发生整数上溢，被减小小于其最小值时会发生整数下溢。带符号和无符号的数都有可能发生溢出。

示例 1：有符号和无符号整数的上溢和下溢

```
int i;
unsigned
int j;
i = INT_MAX; // 2,147,483,647
i++;
printf("i = %d\n", i);  // i=-2,147,483,648
```

```
    j = UINT_MAX; // 4,294,967,295;
    j++;
    printf("j = %u\n", j);  // j = 0
    i = INT_MIN; // -2,147,483,648;
    i--;
    printf("i = %d\n", i);  // i = 2,147,483,647
    j = 0; j--;
printf("j = %u\n", j);  // j = 4,294,967,295
```

示例 2：整数下溢导致报文长度异常

```
    /* 报文长度减去 FSM 头的长度*/
     unsigned int length;
    length -= FSM_HDRLEN
```

处理过短报文时，length 的长度可能小于 FSM_HDRLEN，减法的结果小于。由于 length 是无符号数，结果返回了一个很大的数。

正确写法：增加长度检查

```
    if (length < FSM_HDRLEN )
    {
       return VOS_ERROR;
    }
       length -= FSM_HDRLEN;
```

3.12.2.2避免符号错误

有时从带符号整型转换到无符号整型会发生符号错误，符号错误并不丢失数据，但数据失去了原来的含义。 带符号整型转换到无符号整型，最高位（high-order bit）会丧失其作为符号位的功能。如果该带符号整数的值非负，那么转换后值不变；如果该带符号整数的值为负，那么转换后的结果通常是一个非常大的正数。

示例：符号错误绕过长度检查

```
    #define BUF_SIZE 10
```

```
int main(int argc,char* argv[])
{
   int length;
   char buf[BUF_SIZE];
   if (argc != 3)
   {
      return -1;
   }
   length = atoi(argv[1]); //如果 atoi 返回的长度为负数
   if (length < BUF_SIZE)  // len 为负数，长度检查无效
   {
      memcpy(buf, argv[2], length); /* 带符号的 len 被转换为 size_t 类型的无符号整数，负值被解释为一个极大的正整数。memcpy()调用时引发 buf 缓冲区溢出 */
      printf("Data copied\n");
   }
   else
   {
      printf("Too many data\n");
   }
}
```

正确写法 1：将 len 声明为无符号整型

```
#define BUF_SIZE 10
int main(int argc, char* argv[])
{
   unsigned int length;
   char buf[BUF_SIZE];
   if (argc != 3)
```

```
    {
        return -1;
    }
    length = atoi(argv[1]);
    if (length < BUF_SIZE)
    {
        memcpy(buf, argv[2], length);
        printf("Data copied\n");
        }
        else
        {
        printf("Too much data\n");
        }
        return 0;
}
```

正确写法 2：增加对 len 的更有效的范围校验

```
#define BUF_SIZE 10
int main(int argc, char* argv[])
{
    int length;
    char buf[BUF_SIZE];
    if (argc != 3)
{
        return -1;
}
    length = atoi(argv[1]);
    if ((length > 0)  && (length < BUF_SIZE))
        {
```

```
        memcpy(buf, argv[2], length);
        printf("Data copied\n");
        }
    else
    {
        printf("Too much data\n");
    }
    return 0;
    }
```

3.12.2.3避免截断错误

将一个较大整型转换为较小整型，并且该数的原值超出较小类型的表示范围，就会发生截断错误，原值的低位被保留而高位被丢弃。截断错误会引起数据丢失。 使用截断后的变量进行内存操作，很可能会引发问题。

示例：

```
int main(int argc, char* argv[])
{
    unsigned short total = strlen(argv[1]) + strlen(argv[2]) + 1;
    char* buffer = (char*)malloc(total);
    strcpy(buffer, argv[1]);
    strcat(buffer, argv[2]);
    free(buffer);
     return 0;
}
```

示例代码中 total 被定义为 unsigned short，相对于 strlen()的返回值类型 size_t（通常为 unsigned long）太小。如果攻击者提供的两个入参长度分别为 65500 和 36，unsigned long 的 65500+36+1 会被取模截断，total 的最终值是（65500+36+1）%65536 = 1。malloc()只为 buff 分配了 1 字节空间，为 strcpy()和 strcat()的调用创造了缓冲区溢出的条件。

正确写法：将涉及到计算的变量声明为统一的类型，并检查计算结果。

```
int main(int argc, char* argv[])
{
    size_t total = strlen(argv[1]) + strlen(argv[2]) + 1;
    if ((total <= strlen(argv[1])) || (total <= strlen(argv[2])))
    {
        /* handle error */
        return -1;
    }
    char* buffer = (char*)malloc(total);
    strcpy(buffer, argv[1]);
    strcat(buffer, argv[2]);
    free(buffer);
    return 0;
}
```

3.12.3格式化输出安全

3.12.3.1格式字符和参数匹配

使用格式化字符串应该小心，确保格式字符和参数之间的匹配，保留数量和数据类型。格式字符和参数之间的不匹配会导致未定义的行为。大多数情况下，不正确的格式化字符串会导致程序异常终止。

示例：

```
char *error_msg = "Resource not available to user.";
int error_type = 3;
/* 格式字符和参数的类型不匹配*/
printf("Error (type %s): %d\n", error_type, error_msg);
/* 格式字符和参数的数量不匹配*/
printf("Error: %s\n");
```

格式化字符串在编码时会大量使用，容易 copy-paste 省事，这就容易出现不匹配的错误。

3.12.3.2用户输入格式化

避免将用户输入作为格式化字符串的一部分或者全部。调用格式化 I/O 函数时，不要直接或者间接将用户输入作为格式化字符串的一部分或者全部。攻击者对一个格式化字符串拥有部分或完全控制，存在以下风险：进程崩溃、查看栈的内容、改写内存、甚至执行任意代码。

示例 1：

```
char input[1000];
if (fgets(input, sizeof(input) - 1, stdin) == NULL)
{
   /* handle error */
}
   input[sizeof(input)-1] = '\0';
   printf(input);
```

上述代码 input 直接来自用户输入，并作为格式化字符串直接传递给 printf()。当用户输入的是“%s%s%s%s%s%s%s%s%s%s%s”，就可能触发无效指针或未映射的地址读取。格式字符%s 显示栈上相应参数所指定的地址的内存。这里 input 被当成格式化字符串，而没有提供参数，因此 printf()读取栈中任意内存位置，指导格式字符耗尽或者遇到一个无效指针或未映射地址为止。

正确做法：给 printf()传两个参数，第一个参数为”%s”，目的是将格式化字符串确定下来；第二个参数为用户输入 input。

```
char input[1000];
if (fgets(input, sizeof(input)-1, stdin) == NULL)
{
   /* handle error */
}
input[sizeof(input)-1] = '\0';
printf(“%s”, input);
```

示例 2：

```
void check_password(char *user, char *password)
```

```
{
   if (strcmp(password(user), password) != 0)
      {
         char *msg = malloc(strlen(user) + 100);
        if (!msg)
         {
            /* handle error condition */
         }
         sprintf(msg, "%s login incorrect", user);
         fprintf(STDERR, msg);
         syslog(LOG_INFO, msg);
         free(msg);
      }
   /*…*/
}
```

上述代码检查给定用户名及其口令是否匹配，当不匹配时显示一条错误信息，并将错误信息写入日志中。同样的，如果 user 为”%s%s%s%s%s%s%s%s%s%s%s%s”，经过格式化函数 sprintf()的拼装后，msg 指向的字符串为”%s%s%s%s%s%s%s%s%s%s%s%s login incorrect”，在 fprintf()调用中，msg 将作为 fprintf()的格式化字符串，可能引发如同示例 1 一样的问题。而且，syslog()函数也一样存在格式化字符串的问题。

正确做法：格式化字符串由代码确定，未经检查过滤的用户输入只能作为参数。

```
void check_password(char *user, char *password)
{
   if (strcmp(password(user), password) != 0)
{
      char *msg = malloc(strlen(user) + 100);
       if (!msg)
      {
          /* handle error condition */
```

```
        }
        sprintf(msg, "%s password incorrect", user);
        fprintf(stderr, "%s", user);
        syslog(LOG_INFO, "%s", msg);
        free(msg);
        }
    }
```

3.12.4文件 I/O 安全

3.12.4.1禁止使用 strlen()计算二进制数据的长度

strlen()函数用于计算字符串的长度，它返回字符串中第一个 NULL 结束符之前的字符的数量。因此用 strlen()处理文件 I/O 函数读取的内容时要小心，因为这些内容可能是二进制也可能是文本。

示例：

```
char buf[BUF_SIZE + 1];
if (fgets(buf, sizeof(buf), fp) == NULL)
{
   /* handle error */
}
buf[strlen(buf) - 1] = '\0';
```

上述代码试图从一个输入行中删除行尾的换行符（\n）。如果 buf 的第一个字符是 NULL，strlen(buf)返回 0，这时对 buf 进行数组下标为[-1]的访问操作将会越界。

正确做法：在不能确定从文件读取到的数据的类型时，不要使用依赖 NULL 结束符的字符串操作函数。

```
 char buf[BUF_SIZE + 1];
char *p;
if (fgets(buf, sizeof(buf), fp))
{
  p = strchr(buf, '\n');
```

```
 if (p)
      {
      *p = '\0';
      }
   }
else
{
   /* handle error condition */
}
```

3.12.4.2int 类型接受字符 I/O 返回值

字符 I/O 函数 fgetc()、getc()和 getchar()都从一个流读取一个字符，并把它以 int 值的形式返回。如果这个流到达了文件尾或者发生读取错误，函数返回 EOF。fputc()、putc()、putchar()和 ungetc()也返回一个字符或 EOF。

如果这些 I/O 函数的返回值需要与 EOF 进行比较，不要将返回值转换为 char 类型。因为 char 是有符号 8 位的值，int 是 32 位的值。如果 getchar()返回的字符的 ASCII 值为 0xFF，转换为 char 类型后将被解释为 EOF。因为这个值被有符号扩展为 0xFFFFFFFF（EOF 的值）执行比较。

示例：

```
char buf[BUF_SIZE];
char ch;
int i = 0;
while ( (ch = getchar()) != '\n' && ch != EOF )
{
   if ( i < BUF_SIZE - 1 )
   {
      buf[i++] = ch;
   }
}
buf[i] = '\0'; /* terminate NTBS */
```

正确做法：使用 int 类型的变量接受 getchar()的返回值。

```
char buf[BUF_SIZE];
int ch;
int i = 0;
while (((ch = getchar()) != '\n') && ch != EOF)
{
   if (i < BUF_SIZE - 1)
   {
      buf[i++] = ch;
   }
}
buf[i] = '\0'; /* terminate NTBS */
```

对于 sizeof(int) == sizeof(char)的平台，用 int 接收返回值也可能无法与 EOF 区分，这时要用 feof()和 ferror()检测文件尾和文件错误。

3.13 其他

3.13.1防止命令注入。

C99 函数 system()通过调用一个系统定义的命令解析器（如 UNIX 的 shell，Windows 的 CMD.exe）来执行一个指定的程序/命令。类似的还有 POSIX 的函数 popen()。

如果 system()的参数由用户的输入组成，恶意用户可以通过构造恶意输入，改变 system()调用的行为。

示例：

system(sprintf("any_exe %s", input));

如果恶意用户输入参数：

happy; useradd attacker

最终 shell 将字符串“any_exe happy; useradd attacker”解释为两条独立的命令：

正确做法：使用 POSIX 函数 execve()代替 system().

```
void secuExec (char *input)
{
   pid_t pid;
   char *const args[] = {"", input, NULL};
   char *const envs[] = {NULL};
   pid = fork();
   if (pid == -1)
      {
       puts("fork error");
      }
   else if (pid == 0)
      {
         if (execve("/usr/bin/any_exe", args, envs) == -1)
         {
            puts("Error executing any_exe");
            }
      }
  return;
}
```

Windows 环境可能对 execve()的支持不是很完善，建议使用 Win32 API CreateProcess()代替 system()