文档名称：

C 和 C++安全编程指南

摘 要：

介绍天津华宁电子研发中心关于 C 和 C++编程语言的设计规范，帮助软件工程师编写安全可靠的应用程序。

当前版本	V1.0	文件状态	[]草稿; [√]正式	取代版本	V1.0
完成时间	2023-12-25		被取代文档完成时间		
作者	刘忠义		审批		
批准			文档编号	HN/WI-GC-YF-061	
版本历史：					
版本状态	作者	参与者	完成日期	备注	
V1.0	刘忠义		2023-12		

目 录

1、引言

1.1 编写目的

为了提高产品代码质量，方便软件开发人员编写出简洁，高效，移植性高的代码，结合《华为 C&C++安全指南》，《Google C++ Style Guide》，《腾讯代码安全指南》等安全代码编写规范说明，并结合公司编码规范编写此文档。

1.2 参考资料

《华为 C&C++安全指南》

参考地址：由于不是以网站形式发布的，这里就不提供链接了。

附件：

华为C&C 语言安全编程规范_V3.1.pdf

《Google C++ Style Guide》

参考地址：https://github.com/zh-google-styleguide/zh-google-styleguide

附件：

Google C++ Style Guide.pdf

《腾讯代码安全指南》

参考地址：https://github.com/Tencent/secguide

附件：

腾讯代码安全指南.pdf

《360 代码安全指南》

参考地址：https://github.com/Qihoo360/safe-rules

360代码安全指南.pdf

附件：

《C++代码规范》

C++语言编程规范.xlsx

附件：

《C 语言代码规范》

C语言编程规范.xlsx

附件：

2、通用规范

2.1 清晰第一

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

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

“编写程序应该以人为本，计算机第二。”——Steve McConnell

本规范通过后文中的原则（如头优秀的代码可以自我解释，不通过注释即可轻易读懂/头文件中适合放置接口的声明，不适合放置实现/除了常见的通用缩写以外，不使用单词缩写，不得使用汉语拼音）、规则（如防止局部变量与全局变量同名）等说明清晰的重要性。

一般情况下，代码的可阅读性高于性能，只有确定性能是瓶颈时，才应该主动优化。

2.2 简洁为美

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

废弃的代码(没有被调用的函数和全局变量)要及时清除，重复代码应该尽可能提炼成函数。

本规范通过后文中的原则（如文件应当职责单一/一个函数仅完成一件功能）、规则（重复代码应该尽可能提炼成函数/避免函数过长，新增函数不超过 50 行）等说明简洁的重要性。

2.3 保持风格统一

产品所有人共同分享同一种风格所带来的好处，远远超出为了统一而付出的代价。在公司已有编码规范的指导下，审慎地编排代码以使代码尽可能清晰，是一项非常重要的技能。如果重构/修改其他风格的代码时，比较明智的做法是根据现有代码的现有风格继续编写代码，或者使用格式转换工具进行转换成公司内部风格。

3、规范实施

本文是适用于不同应用场景的规则集合，读者可选取适合自己需求的建议和规则。

指出某种错误的规则，如有“不可”、“不应”等字样的规则应尽量被选取，有“禁用”等字样的规则可能只适用于某一场景，可酌情选取

4、术语定义

规则包括：

编号：规则在本文中的章节编号

名称：用简练的短语描述违反规则的状况

标题：规则的定义

说明：规则设立的原因、违反规则的后果、示例、改进建议、参照依据、参考资料等内容

5、代码安全准则

5.1 头文件

不合理的头文件布局是编译时间过长的根因，不合理的头文件实际上反映了不合理的设计。

（1） 头文件中适合放置接口的声明，不适合放置实现

头文件是模块（Module）或单元（Unit）的对外接口。头文件中应放置对外部的声明，如对外提供

的函数声明、宏定义、类型定义等。

要求：

内部使用的函数（相当于类的私有方法）声明不应放在头文件中。

内部使用的宏、枚举、结构定义不应放入头文件中。

变量定义不应放在头文件中，应放在.c 文件中。

变量的声明尽量不要放在头文件中，亦即尽量不要使用全局变量作为接口。变量是模块或单元的内部实现细节，不应通过在头文件中声明的方式直接暴露给外部，应通过函数接口的方式进行对外暴露。即使必须使用全局变量，也只应当在.c 中定义全局变量，在.h 中仅声明变量为全局的。

（2） 头文件应当职责单一，切忌依赖复杂

头文件过于复杂，依赖过于复杂是导致编译时间过长的主要原因。很多现有代码中头文件过大，职责过多，再加上循环依赖的问题，可能导致为了在.c 中使用一个宏，而包含十几个头文件。

（3） 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。

（4） 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。

（5）.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。

（6）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。

（7） 禁止在头文件中定义变量

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。

（8） #include 的路径及顺序

使用标准的头文件包含顺序可增强可读性, 避免隐藏依赖: 相关头文件, C 库, C++ 库, 其他库的 .h, 本项目内的 .h.

如, dir/foo.cc 或 dir/foo_test.cc 的主要作用是实现或测试 dir2/foo2.h 的功能, foo.cc 中包含头文件的次序如下:

dir2/foo2.h (优先位置, 详情如下)

C 系统文件

C++ 系统文件

其他库的 .h 文件

本项目内 .h 文件

例外：

有时，平台特定（system-specific）代码需要条件编译（conditional includes），这些代码可以放到其它 includes 之后。当然，您的平台特定代码也要够简练且独立，比如：

```
#include "foo/public/fooserver.h"
#include "base/port.h"  // For LANG_CXX11.
#ifdef LANG_CXX11#include <initializer_list>#endif  // LANG_CXX11
```

5.2 函数

函数设计的精髓：编写整洁函数，同时把代码有效组织起来。

整洁函数要求：代码简单直接、不隐藏设计者的意图、用干净利落的抽象和直截了当的控制语句将函数有机组织起来。

代码的有效组织包括：逻辑层组织和物理层组织两个方面。逻辑层，主要是把不同功能的函数通过某种联系组织起来，主要关注模块间的接口，也就是模块的架构。物理层，无论使用什么样的目录或者

名字空间等，需要把函数用一种标准的方法组织起来。例如：设计良好的目录结构、函数名字、文件组织等，这样可以方便查找。

（1）一个函数仅完成一件功能

一个函数实现多个功能给开发、使用、维护都带来很大的困难。

将没有关联或者关联很弱的语句放到同一函数中，会导致函数职责不明确，难以理解，难以测试和改动。

（2）重复代码应该尽可能提炼成函数

重复代码提炼成函数可以带来维护成本的降低。

重复代码是我司不良代码最典型的特征之一。在“代码能用就不改”的指导原则之下，大量的烟囱式设计及其实现充斥着各产品代码之中。新需求增加带来的代码拷贝和修改，随着时间的迁移，产品中堆砌着许多类似或者重复的代码。

项目组应当使用代码重复度检查工具，在持续集成环境中持续检查代码重复度指标变化趋势，并对新增重复代码及时重构。当一段代码重复两次时，即应考虑消除重复，当代码重复超过三次时，应当立刻着手消除重复。

（3）避免函数过长，新增函数不超过 50 行 （非空非注释行）

过长的函数往往意味着函数功能不单一，过于复杂。

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。

（4）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

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。

（5） 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;
}
```

（6）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。

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。

（7）对函数的错误返回码要全面处理

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

（8）设计高扇入，合理扇出（小于 7）的函数

扇出是指一个函数直接调用（控制）其它函数的数目，而扇入是指有多少上级函数调用它。

扇出过大，表明函数过分复杂，需要控制和协调过多的下级函数；而扇出过小，例如：总是 1，表明函数的调用层次可能过多，这样不利于程序阅读和函数结构的分析，并且程序运行时会对系统资源如堆栈空间等造成压力。通常函数比较合理的扇出（调度函数除外）通常是 3~5。

扇出太大，一般是由于缺乏中间层次，可适当增加中间层次的函数。扇出太小，可把下级函数进一步分解多个函数，或合并到上级函数中。当然分解或合并函数时，不能改变要实现的功能，也不能违背函数间的独立性。扇入越大，表明使用此函数的上级函数越多，这样的函数使用效率高，但不能违背函数间的独立性而单纯地追求高扇入。公共模块中的函数及底层函数应该有较高的扇入。

较良好的软件结构通常是顶层函数的扇出较高，中层函数的扇出较少，而底层函数则扇入到公共模块中。

（9）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。

（10）函数不变参数使用 const

不变的值更易于理解/跟踪和分析，把 const 作为默认选项，在编译时会对其进行检查，使代码更牢固/更安全。

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

```
int Strncmp(const char *s1, const char *s2, register size_t n)
{
    register unsigned char u1, u2;
    while (n-- > 0)
    { u1 = (unsigned char) *s1++;u2 = (unsigned char) *s2++;
        if (u1 != u2)
        {
            return u1 - u2;
        }
        if (u1 == '\0')
        {
            return 0;
        }
    }
```

```
    return 0;
}
```

（11）函数应避免使用全局变量、静态局部变量和 I/O 操作，不可避免的地方应集中使用

带有内部“存储器”的函数的功能可能是不可预测的，因为它的输出可能取决于内部存储器（如某标记）的状态。这样的函数既不易于理解又不利于测试和维护。在 C 语言中，函数的 static 局部变量是函数的内部存储器，有可能使函数的功能不可预测。

错误示例：如下函数，其返回值（即功能）是不可预测的。

```
unsigned int IntegerSum( unsigned int base ){
    unsigned int index;
    static unsigned int sum = 0;// 注意，是 static 类型的。
    // 若改为 auto 类型，则函数即变为可预测。
    for (index = 1; index <= base; index++)
    {
        sum += index;
    }
    return sum;
}
```

（12）检查函数所有非参数输入的有效性，如数据文件、公共变量等

函数的输入主要有两种：一种是参数输入；另一种是全局变量、数据文件的输入，即非参数输入。函数在使用输入参数之前，应进行有效性检查。

（13） 函数的参数个数不超过 5 个

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

函数的参数个数不要超过 5 个，如果超过了建议拆分为不同函数。

（14）除打印类函数外，不要使用可变长参函数。

可变长参函数的处理过程比较复杂容易引入错误，而且性能也比较低，使用过多的可变长参函数将导致函数的维护难度大大增加。

（15）在源文件范围内声明和定义的所有函数，除非外部可见，否则应该增加 static 关键字

如果一个函数只是在同一文件中的其他地方调用，那么就用 static 声明。使用 static 确保只是在声明它的文件中是可见的，并且避免了和其他文件或库中的相同标识符发生混淆的可能性。

正确示例：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，以便于后续的打热补丁等操作。

```
#ifdef _DEBUG

#define STATIC static

#else

#define STATIC

#endif
```

（16）引用参数函数重载

在 C 语言中, 如果函数需要修改变量的值, 参数必须为指针, 如 ``int foo(int *pval)``. 在 C++ 中, 函数还可以声明为引用参数: ``int foo(int &val)``.

事实上这在 Google Code 是一个硬性约定: 输入参数是值参或 ``const`` 引用, 输出参数为指针. 输入参数可以是 ``const`` 指针, 但决不能是非 ``const`` 的引用参数, 除非特殊要求, 比如 ``swap()``.

有时候, 在输入形参中用 ``const T*`` 指针比 ``const T&`` 更明智. 比如:

* 可能会传递空指针.

* 函数要把指针或对地址的引用赋值给输入形参.

总而言之, 大多时候输入形参往往是 ``const T&``. 若用 ``const T*`` 则说明输入另有处理. 所以若要使用 ``const T*``, 则应给出相应的理由, 否则会使得读者感到迷惑.

（17）函数重载

如果打算重载一个函数, 可以试试改在函数名里加上参数信息. 例如, 用 ``AppendString()`` 和 ``AppendInt()`` 等, 而不是一口气重载多个 ``Append()``. 如果重载函数的目的是为了支持不同数量的同一类型参数, 则优先考虑使用 ``std::vector`` 以便使用者可以用 :ref:`列表初始化 <braced-initializer-list>` 指定参数。

5.3 标识符命名与定义

标识符的命名规则历来是一个敏感话题，典型的命名风格如 unix 风格、windows 风格等，从来无法达成共识。实际上，各种风格都有其优势也有其劣势，而且往往和个人的审美观有关。我们对标识符定义主要是为了让团队的代码看起来尽可能统一，有利于代码的后续阅读和修改，产品可以根据自己的实际需要指定命名风格，规范中不再做统一的规定。

这部分详情见《C 语言代码规范》，《C++代码规范》。

5.4 变量

（1）一个变量只有一个功能，不能把一个变量用作多种用途

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。

错误示例：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 Ret;
    }
```

（2）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。

错误示例：如下结构不太清晰、合理。

```
    typedef struct STUDENTSTRU
    {
        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;
```

正确示例：若改为如下，会更合理些。

```
typedef struct TEACHERSTRU
{
    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 STUDENTSTRU
{
    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）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。

（4）防止局部变量与全局变量同名

尽管局部变量和全局变量的作用域不同而不会发生语法错误，但容易使人误解。

（5）通讯过程中使用的结构，必须注意字节序

通讯报文中，字节序是一个重要的问题，我司设备使用的 CPU 类型复杂多样，大小端、32 位/64 位的处理器也都有，如果结构会在报文交互过程中使用，必须考虑字节序问题。由于位域在不同字节序下，表现看起来差别更大，所以更需要注意对于这种跨平台的交互，数据成员发送前，都应该进行主机序到网络序的转换；接收时，也必须进行网络序到主机序的转换。

（6）严禁使用未经初始化的变量作为右值

在首次使用前初始化变量，初始化的地方离使用的地方越近越好。

（7）构造仅有一个模块或函数可以修改、创建，而其余有关模块或函数只访问的全局变量，防止多个不同模块或函数都可以修改、创建同一全局变量的现象

降低全局变量耦合度。

（8）使用面向接口编程思想，通过 API 访问数据：如果本模块的数据需要对外部模块开放 ，应提供接口函数来设置、获取，同时注意全局数据的访问互斥

避免直接暴露内部数据给外部模型使用，是防止模块间耦合最简单有效的方法。定义的接口应该有比较明确的意义，比如一个风扇管理功能模块，有自动和手动工作模式，那么设置、查询工作模块就可以定义接口为 SetFanWorkMode，GetFanWorkMode；查询转速就可以定义为 GetFanSpeed；风扇支持节能功能开关，可以定义 EnabletFanSavePower 等。

（9）明确全局变量的初始化顺序，避免跨模块的初始化依赖

系统启动阶段，使用全局变量前，要考虑到该全局变量在什么时候初始化，使用全局变量和初始化全局变量，两者之间的时序关系，谁先谁后，一定要分析清楚，不然后果往往是低级而又灾难性的。

（10）尽量减少没有必要的数据类型默认转换与强制转换

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

错误示例：如下赋值，多数编译器不产生告警，但值的含义还是稍有变化。

```
char ch;

unsigned short int exam;

ch = -1;

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

5.5 注释

注释虽然写起来很痛苦, 但对保证代码可读性至关重要. 下面的规则描述了如何注释以及在哪儿注释. 当然也要记住: 注释固然很重要, 但最好的代码应当本身就是文档. 有意义的类型名和变量名, 要远胜过要用注释解释的含糊不清的名字.

你写的注释是给代码读者看的, 也就是下一个需要理解你的代码的人. 所以慷慨些吧, 下一个读者可能就是你!

（1）注释风格

使用 ``//`` 或 ``/* */``, 统一就好.

详情请见《C 语言代码规范》，《C++代码规范》

（2）文件注释

在每一个文件开头加入版权公告.

文件注释描述了该文件的内容. 如果一个文件只声明, 或实现, 或测试了一个对象, 并且这个对象已经在它的声明处进行了详细的注释, 那么就没必要再加上文件注释. 除此之外的其他文件都需要文件注释.

详情请见《C 语言代码规范》，《C++代码规范》

（3）类注释

每个类的定义都要附带一份注释, 描述类的功能和用法, 除非它的功能相当明显.

详情请见《C 语言代码规范》，《C++代码规范》

（4）函数注释

函数声明处的注释描述函数功能; 定义处的注释描述函数实现。

基本上每个函数声明处前都应当加上注释, 描述函数的功能和用途。只有在函数的功能简单而明显时才能省略这些注释(例如, 简单的取值和设值函数). 注释使用叙述式 ("Opens the file") 而非指令式 ("Open the file"); 注释只是为了描述函数, 而不是命令函数做什么. 通常, 注释不会描述函数如何工作.。那是函数定义部分的事情。

详情请见《C 语言代码规范》，《C++代码规范》

（5）变量注释

通常变量名本身足以很好说明变量用途. 某些情况下, 也需要额外的注释说明。

每个类数据成员 (也叫实例变量或成员变量) 都应该用注释说明用途. 如果有非变量的参数(例如特殊值, 数据成员之间的关系, 生命周期等)不能够用类型与变量名明确表达, 则应当加上注释. 然而, 如果变量类型与变量名已经足以描述一个变量, 那么就不再需要加上注释。

特别地, 如果变量可以接受 ``NULL`` 或 ``-1`` 等警戒值, 须加以说明。

详情请见《C 语言代码规范》，《C++代码规范》

（6） 行注释

比较隐晦的地方要在行尾加入注释. 在行尾空两格进行注释。

详情请见《C 语言代码规范》，《C++代码规范》

（7） 函数参数注释

如果函数参数的意义不明显, 考虑用下面的方式进行弥补:

- 如果参数是一个字面常量, 并且这一常量在多处函数调用中被使用, 用以推断它们一致, 你应当用一个常量名让这一约定变得更明显, 并且保证这一约定不会被打破.

- 考虑更改函数的签名, 让某个 ``bool`` 类型的参数变为 ``enum`` 类型, 这样可以让这个参数的值表达其意义.

- 如果某个函数有多个配置选项, 你可以考虑定义一个类或结构体以保存所有的选项, 并传入类或结构体的实例. 这样的方法有许多优点, 例如这样的选项可以在调用处用变量名引用, 这样就能清晰地表明其意义. 同时也减少了函数参数的数量, 使得函数调用更易读也易写. 除此之外, 以这样的方式, 如果你使用其他的选项, 就无需对调用点进行更改.

- 用具名变量代替大段而复杂的嵌套表达式。
- 万不得已时，才考虑在调用点用注释阐明参数的意义。

详情请见《C 语言代码规范》，《C++代码规范》

（8） TODO 注释

对那些临时的，短期的解决方案，或已经够好但仍不完美的代码使用 ``TODO`` 注释.

``TODO`` 注释要使用全大写的字符串 ``TODO``，在随后的圆括号里写上你的名字，邮件地址，bug ID，或其它身份标识和与这一 ``TODO`` 相关的 issue. 主要目的是让添加注释的人 (也是可以请求提供更多细节的人) 可根据规范的 ``TODO`` 格式进行查找. 添加 ``TODO`` 注释并不意味着你要自己来修正，因此当你加上带有姓名的 ``TODO`` 时，一般都是写上自己的名字.

```
// TODO(kl@gmail.com): Use a "*" here for concatenation operator.

// TODO(Zeke) change this to use relations.

// TODO(bug 12345): remove the "Last visitors" feature
```

如果加 ``TODO`` 是为了在 "将来某一天做某事"，可以附上一个非常明确的时间 "Fix by November 2005")，或者一个明确的事项 ("Remove this code when all clients can handle XML responses.").

（9） 弃用注释

通过弃用注释（``DEPRECATED`` comments）以标记某接口点已弃用.

您可以写上包含全大写的 ``DEPRECATED`` 的注释，以标记某接口为弃用状态. 注释可以放在接口声明前，或者同一行.

在 ``DEPRECATED`` 一词后，在括号中留下您的名字，邮箱地址以及其他身份标识.

弃用注释应当包涵简短而清晰的指引，以帮助其他人修复其调用点. 在 C++ 中，你可以将一个弃用函数改造成一个内联函数，这一函数将调用新的接口.

仅仅标记接口为 ``DEPRECATED`` 并不会让大家不约而同地弃用，您还得亲自主动修正调用点（callsites），或是找个帮手.

修正好的代码应该不会再涉及弃用接口点了，着实改用新接口点. 如果您不知从何下手，可以找标记弃用注释的当事人一起商量.

5.6 代码编译

（1） 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

编译器是你的朋友，如果它发出某个告警，这经常说明你的代码中存在潜在的问题。

（2） 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。

（3） 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

（4） 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。

（5） 要小心地使用编辑器提供的块拷贝功能编程

5.7 C/C++通用安全指南

5.7.1 建议

建议 5.7.1.1：字符串或指针作为函数参数时，请检查参数是否为 NULL

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，如果上层调用者已经保证了该参数不可能为 NULL，在调用本函数时，在函数开始处可以加 ASSERT 进行校验。 例如下面的代码，因为 BYTE *p 有可能为 NULL,因此在使用前需要进行判断。

```
int Foo(int *p, int count)
{
    if (p != NULL && count > 0) {
        int c = p[0];
    }
    ...
}
```

```
int Foo2()
{
    int *arr = ...
    int count = ...
    Foo(arr, count);
...
}
```

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

```
int Foo(int *p, int count)
{
    ASSERT(p != NULL); //ASSERT is added to verify p.
    ASSERT(count > 0);
    int c = p[0];
     ...
    }
int Foo2()
{
    int *arr = ...
    int count = ...
    ...
    if (arr != NULL && count > 0) {
```

```
        Foo(arr, count);
    }
    ...
}
```

建议 5.7.1.2：禁用 exit、ExitProcess 函数（main 函数除外）

程序应该安全退出，除了 main 函数以外，禁止任何地方调用 exit、ExitProcess 函数退出进程。直接退出进程会导致代码的复用性降低，资源得不到有效地清理。 程序应该通过错误值传递的机制进行错误处理。 以下代码加载文件，加载过程中如果出错，直接调用 exit 退出：

```
void LoadFile(const char *filePath)
{
    FILE* fp = fopen(filePath, "rt");
    if (fp == NULL) {
        exit(0);
    }
    ...
}
```

正确的做法应该通过错误值传递机制，例如：

```
BOOL LoadFile(const char *filePath)
{
    BOOL ret = FALSE;
    FILE* fp = fopen(filePath, "rt");
    if (fp != NULL) {
        ...
```

```
        }
        ...
        return ret;
    }
```

建议 5.7.1.3：禁用 abort 函数

abort 会导致程序立即退出，资源得不到清理。 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);
            ...
}
```

建议 5.7.1.4 不得直接使用刚分配的未初始化的内存（如 realloc）

一些刚申请的内存通常是直接从堆上分配的，可能包含有旧数据的，直接使用它们而不初始化，可能会导致安全问题。例如，CVE-2019-13751。应确保初始化变量，或者确保未初始化的值不会泄露给用户。

```
// Badchar* Foo() {
    char* a = new char[100];
```

```
    a[99] = '\0';
    memcpy(a, "char", 4);
    return a;
}
// Goodchar* Foo() {
    char* a = new char[100];
    memcpy(a, "char", 4);
    a[4] = '\0';
    return a;
}
```

在 C++ 中，再次强烈推荐用 string、vector 代替手动内存分配。

建议 5.7.1.5：确认 if 里面的按位操作

if 里，非 bool 类型和非 bool 类型的按位操作可能代表代码存在错误。

```
// Badvoid Foo() {
    int bar = 0x1;          // binary 01
    int foobar = 0x2;       // binary 10

    if (foobar & bar)         // result = 00, false
        ...
}
```

上述代码可能应该是：

```
// Goodvoid foo() {
```

```
    int      bar = 0x1;
    int foobar = 0x2;


    if (foobar && bar)    // result : true
        ...
}
```

建议 5.7.1.6：如果类的公共接口中返回类的私有数据地址，则必须加 const 类型

实例：

```
class CMsg {
public:
    CMsg();
    ~CMsg();
    Const unsigned char *GetMsg();
protected:
    int size;
    unsigned char *msg;
};
CMsg::CMsg()
{
    size = 0;
    msg = NULL;
}
```

```
const unsigned char *CMsg::GetMsg()
{
    return msg;
}
```

建议 5.7.1.7：内存申请前，必须对申请内存大小进行合法性校验

内存申请的大小可能来自于外部数据，必须检查其合法性，防止过多地、非法地申请内存。不能申请 0 长度的内存。 例如：

```
int Foo(int size)
{
    if (size <= 0) {
    //error
    ...
    }
    ...
    char *msg = (char *)malloc(size);
    ...
}
```

建议 5.7.1.8：内存分配后必须判断是否成功

```
char *msg = (char *)malloc(size);
if (msg != NULL) {
    ...
}
```

建议 5.7.1.9：禁止使用 realloc()函数

realloc()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。 1） 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； 2） 参数 ptr 为 NULL，但 size 不为 0，那么行为等同于 malloc(size)； 3） 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。虽然在编码中提供了一些便利性，但是却极易引发各种 bug。

建议 5.7.1.10：不得直接使用无长度限制的字符拷贝函数

不应直接使用 legacy 的字符串拷贝、输入函数，如 strcpy、strcat、sprintf、wcscpy、mbscpy 等，这些函数的特征是：可以输出一长串字符串，而不限制长度。如果环境允许，应当使用其_s 安全版本替代，或者使用 n 版本函数（如：snprintf，vsnprintf）。

若使用形如 sscanf 之类的函数时，在处理字符串输入时应当通过%10s 这样的方式来严格限制字符串长度，同时确保字符串末尾有\0。如果环境允许，应当使用_s 安全版本。

使用 n 系列拷贝函数时，要确保正确计算缓冲区长度，同时，如果你不确定是否代码在各个编译器下都能确保末尾有 0 时，建议可以适当增加 1 字节输入缓冲区，并将其置为\0，以保证输出的字符串结尾一定有\0。

一些需要注意的函数，例如 strncpy 和_snprintf 是不安全的。 strncpy 不应当被视为 strcpy 的 n 系列函数，它只是恰巧与其他 n 系列函数名字很像而已。strncpy 在复制时，如果复制的长度超过 n，不会在结尾补\0。

在 C++ 中，强烈建议用 string、vector 等更高封装层次的基础组件代替原始指针和动态数组，对提高代码的可读性和安全性都有很大的帮助。

建议 5.7.1.11：尽量减少使用 _alloca 和可变长度数组

_alloca 和可变长度数组使用的内存量在编译期间不可知。尤其是在循环中使用时，根据编译器的实现不同，可能会导致：（1）栈溢出，即拒绝服务； （2）缺少栈内存测试的编译器实现可能导致申请到非栈内存，并导致内存损坏。这在栈比较小的程序上，例如 IoT 设备固件上影响尤为大。对于 C++，

可变长度数组也属于非标准扩展，在代码规范中禁止使用。

错误示例：

```
// Bad
for (int i = 0; i < 100000; i++) {
    char* foo = (char *)_alloca(0x10000);
    ..do something with foo ..;
}
void Foo(int size) {
    char msg[size]; //  不可控的栈溢出风险！
}
```

正确示例：

```
// Good//  改用动态分配的堆内存
for (int i = 0; i < 100000; i++) {
    char * foo = (char *)malloc(0x10000);
    ..do something with foo ..;
    if (foo_is_no_longer_needed) {
        free(foo);
        foo = NULL;
    }
}
void Foo(int size) {
    std::string msg(size, '\0');    // C++
    char* msg = malloc(size);    // C
```

建议 5.7.1.12：创建进程类的函数的安全规范

system、WinExec、CreateProcess、ShellExecute 等启动进程类的函数，需要严格检查其参数。

启动进程需要加上双引号，错误例子：

// BadWinExec("D:\\program files\\my folder\\foobar.exe", SW_SHOW);

当存在 D:\program files\my.exe 的时候，my.exe 会被启动。而 foobar.exe 不会启动。

// GoodWinExec("\"D:\\program files\\my folder\\foobar.exe\"", SW_SHOW);

另外，如果启动时从用户输入、环境变量读取组合命令行时，还需要注意是否可能存在命令注入。

```
// Bad

std::string cmdline = "calc ";

cmdline += user_input;system(cmdline.c_str());
```

比如，当用户输入 1+1 && ls 时，执行的实际上是 calc 1+1 和 ls 两个命令，导致命令注入。

需要检查用户输入是否含有非法数据。

```
// Good

std::string cmdline = "ls ";

cmdline += user_input;

if(cmdline.find_first_not_of("1234567890.+-*/e ") == std::string::npos)

   system(cmdline.c_str());else

   warning(...);
```

建议 5.7.1.13：printf 系列参数必须对应

所有 printf 系列函数，如 sprintf，snprintf，vprintf 等必须对应控制符号和参数。

错误示例：

```
// Badconst int buf_size = 1000;char buffer_send_to_remote_client[buf_size] = {0};

snprintf(buffer_send_to_remote_client, buf_size, "%d: %p", id, some_string);   // %p 应为 %s

buffer_send_to_remote_client[buf_size - 1] = '\0';
```

```
send_to_remote(buffer_send_to_remote_client);
```

正确示例：

```
// Goodconst int buf_size = 1000;char buffer_send_to_remote_client[buf_size] = {0};
snprintf(buffer_send_to_remote_client, buf_size, "%d: %s", id, some_string);
buffer_send_to_remote_client[buf_size - 1] = '\0';
send_to_remote(buffer_send_to_remote_client);
```

前者可能会让 client 的攻击者获取部分服务器的原始指针地址，可以用于破坏 ASLR 保护。

建议 5.7.1.14：防止泄露指针（包括%p）的值

所有 printf 系列函数，要防止格式化完的字符串泄露程序布局信息。例如，如果将带有%p 的字符串泄露给程序，则可能会破坏 ASLR 的防护效果。使得攻击者更容易攻破程序。

%p 的值只应当在程序内使用，而不应当输出到外部或被外部以某种方式获取。

错误示例：

```
// Bad// 如果这是暴露给客户的一个 API：
uint64_t GetUniqueObjectId(const Foo* pobject) {
  return (uint64_t)pobject;
}
```

正确示例：

```
// Good
uint64_t g_object_id = 0;
void Foo::Foo() {
  this->object_id_ = g_object_id++;
}
// 如果这是暴露给客户的一个 API：
```

```
uint64_t GetUniqueObjectId(const Foo* object) {
  if (object)
    return object->object_id_;
  else
    error(...);
}
```

建议 5.7.1.15：不应当把用户可修改的字符串作为 printf 系列函数的“format”参数

如果用户可以控制字符串，则通过 %n %p 等内容，最坏情况下可以直接执行任意恶意代码。

在以下情况尤其需要注意： WIFI 名，设备名……

错误：

```
snprintf(buf, sizeof(buf), wifi_name);
```

正确：

```
snprinf(buf, sizeof(buf), "%s", wifi_name);
```

建议 5.7.1.16：检查复制粘贴的重复代码（相同代码通常代表错误）

当开发中遇到较长的句子时，如果你选择了复制粘贴语句，请记得检查每一行代码，不要出现上下两句一模一样的情况，这通常代表代码哪里出现了错误：

```
// Badvoid Foobar(SomeStruct& foobase, SomeStruct& foo1, SomeStruct& foo2) {
  foo1.bar = (foo1.bar & 0xffff) | (foobase.base & 0xffff0000);
  foo1.bar = (foo1.bar & 0xffff) | (foobase.base & 0xffff0000);
}
```

如上例，通常可能是：

```
// Goodvoid Foobar(SomeStruct& foobase, SomeStruct& foo1, SomeStruct& foo2) {
  foo1.bar = (foo1.bar & 0xffff) | (foobase.base & 0xffff0000);
```

```
    foo2.bar = (foo2.bar & 0xffff) | (foobase.base & 0xffff0000);
}
```

最好是把重复的代码片段提取成函数，如果函数比较短，可以考虑定义为 inline 函数，在减少冗余的同时也能确保不会影响性能。

建议 5.7.1.17：函数每个分支都应有返回值

函数的每个分支都应该有返回值，否则如果函数走到无返回值的分支，其结果是未知的。

```
// Badint Foo(int bar) {
    if (bar > 100) {
        return 10;
    } else if (bar > 10) {
        return 1;
    }
}
```

上述例子当 bar<10 时，其结果是未知的值。

```
// Goodint Foo(int bar) {
    if (bar > 100) {
        return 10;
    } else if (bar > 10) {
        return 1;
    }
    return 0;
}
```

开启适当级别的警告（GCC 中为 -Wreturn-type 并已包含在 -Wall 中）并设置为错误，可以在编译阶段发现这类错误。

建议 5.7.1.18：校验内存相关函数的返回值

与内存分配相关的函数需要检查其返回值是否正确，以防导致程序崩溃或逻辑错误。

```
// Badvoid Foo() {
    char* bar = mmap(0, 0x800000, .....);
    *(bar + 0x400000) = '\x88'; // Wrong
}
```

如上例 mmap 如果失败，bar 的值将是 0xffffffff (ffffffff)，第二行将会往 0x3ffffff 写入字符，导致越界写。

```
// Goodvoid Foo() {
    char* bar = mmap(0, 0x800000, .....);
    if(bar == MAP_FAILED) {
        return;
    }

    *(bar + 0x400000) = '\x88';
}
```

建议 5.7.1.19：rand() 类函数应正确初始化

rand 类函数的随机性并不高。而且在使用前需要使用 srand()来初始化。未初始化的随机数可能导致某些内容可预测。

```
// Badint main() {
    int foo = rand();
```

```
    return 0;
}
```

上述代码执行完成后，foo 的值是固定的。它等效于 srand(1); rand()。

```
// Good
int main() {
    srand(time(0));
    int foo = rand();
    return 0;
}
```

建议 5.7.1.20：自己实现的 rand 范围不应过小

如果在弱安全场景相关的算法中自己实现了 PRNG，请确保 rand 出来的随机数不会很小或可预测。

```
// Bad
int32_t val = ((state[0] * 1103515245U) + 12345U) & 999999;
```

上述例子可能想生成 0~999999 共 100 万种可能的随机数，但是 999999 的二进制是 11110100001000111111，与&运算后，0 位一直是 0，所以生成出的范围明显会小于 100 万种。

```
// Good
int32_t val = ((state[0] * 1103515245U) + 12345U) % 1000000;
// Good
int32_t val = ((state[0] * 1103515245U) + 12345U) & 0x7fffffff;
```

建议 5.7.1.21：switch 中应有 default

switch 中应该有 default，以处理各种预期外的情况。这可以确保 switch 接受用户输入，或者后期在其他开发者修改函数后确保 switch 仍可以覆盖到所有情况，并确保逻辑正常运行。

```
// Bad
int Foo(int bar) {
  switch (bar & 7) {
    case 0:
      return Foobar(bar);
      break;
    case 1:
      return Foobar(bar * 2);
      break;
  }}
```

例如上述代码 switch 的取值可能从 0～7，所以应当有 default：

```
// Good
int Foo(int bar) {
  switch (bar & 7) {
    case 0:
      return Foobar(bar);
      break;
    case 1:
      return Foobar(bar * 2);
      break;
    default:
      return -1;
  }}
```

建议 5.7.1.22：仅当只有数据成员时使用 struct，其它一概使用 class.

在 C++ 中 ``struct`` 和 ``class`` 关键字几乎含义一样. 我们为这两个关键字添加我们自己的语义理解, 以便为定义的数据类型选择合适的关键字.

``struct`` 用来定义包含数据的被动式对象, 也可以包含相关的常量, 但除了存取数据成员之外, 没有别的函数功能. 并且存取功能是通过直接访问位域, 而非函数调用. 除了构造函数, 析构函数, ``Initialize()``, ``Reset()``, ``Validate()`` 等类似的用于设定数据成员的函数外, 不能提供其它功能的函数.

如果需要更多的函数功能, ``class`` 更适合. 如果拿不准, 就用 ``class``.

为了和 STL 保持一致, 对于仿函数等特性可以不用 ``class`` 而是使用 ``struct``.

建议 5.7.1.23：声明顺序

将相似的声明放在一起, 将 ``public`` 部分放在最前.

类定义一般应以 ``public:`` 开始, 后跟 ``protected:``, 最后是 ``private:``. 省略空部分.

在各个部分中, 建议将类似的声明放在一起, 并且建议以如下的顺序: 类型 (包括 ``typedef``, ``using`` 和嵌套的结构体与类), 常量, 工厂函数, 构造函数, 赋值运算符, 析构函数, 其它函数, 数据成员。

不要将大段的函数定义内联在类定义中. 通常，只有那些普通的, 或性能关键且短小的函数可以内联在类定义。

建议 5.7.1.24：默认使用 vector，否则选择其他合适的容器

摘要：如果有理由选择某个容器，那么就选择它。如果没有什么特别理由，那么直接选择 vector 即可。

vector 具有很多优点：

容器中最低空间开销。

所有容器中对存放元素存取最快。

与生俱来的数据局部性。

C 兼容内存布局。

迭代器最灵活：随机访问。

性能最高迭代器：指针或性能相当的类。

如果有了理由选择其他容器，那么也没有必要考虑 vector 了，否则无脑 vector 即可。

建议 5.7.1.25：多用范围操作，少用单元素操作

摘要：调用范围操作通常比循环调用单元素操作更加高效和易读。

建议 5.7.1.26：使用 vector 和 string::c_str 与非 C++API 交互

摘要：vector 和 string 的元素内存保证连续，与非 C++API 交互时应该使用他们（如 C）。

对于 vector，可以使用&*v.begin() &v[0] &v.front() vec.data()来获取首元素地址。

对于 string 可以使用 s.c_str() s.data()获取首字符指针。

C++17 起可以直接统一为 std::data(obj)。

建议 5.7.1.27：用 vecotr 和 string 代替数组

几乎同等性能的前提下，提供了更高层次的抽象，自动管理内存，丰富接口，有助于优化。

建议 5.7.1.28：用 push_back 代替其他扩充序列的方式

摘要：尽量使用 push_back。

如果不需要顺序，那么就应该使用 push_back，如果很关心顺序，那么大概不应该选择 vector。

可以通过 back_inserter 配合标准算法使用 push_back。

例外：如果插入范围，那么应该使用 insert。

建议 5.7.1.29：使用公用惯用法真正压缩容量，真正删除元素

摘要：压缩容量，可以使用 swap 惯用法。真正删除元素，可以使用 erase-remove 惯用法。

压缩容量：现已有 shrink_to_fit。

去掉多余容量：container<T>(c).swap(c)。

去掉全部内容和容量：container<T>(c).swap(c)。

删除元素：c.erase(std::remove(c.begin(), c.end(), value), c.end())。

建议 5.7.1.30：尽量减少全局和共享数据

摘要：避免共享数据，尤其是全局数据。共享数据会增加耦合，降低可维护性，通常还会降低性能。

因为使用共享数据的代码片段不仅取决于数据变化的过程，还取决于以后会使用该数据的未知代码

区域的机能。

不同编译单元中全局对象的初始化顺序还是不确定的，应尽量避免使用。

全局的数据还会降低多线程和多处理器环境下的并行性。

即使要用也应该使用工厂来注册与维护。

建议 5.7.1.31：编程中应知道何时与如何考虑可伸缩性

摘要：小心算法复杂度的爆炸增长，不要进行不成熟的提前优化，但是任何时候都应该密切关注算法的复杂度。

通过保证复杂度来保证对未来可能面对的更大数据量下的性能。

即使可预见的未来不会有特别大的数据量，也应该避免不能很好应付数据量增加的算法（除非这种算法确实过于清晰、简单、可读性强）。

一些具体做法：

使用灵活的动态分配的数组，而不是固定大小数组。

了解算法的复杂度。

优先使用快的算法，有对数复杂度就不用线性复杂度，比如能二分查找就绝对不按顺序遍历查找。

即使要优化，也应该尝试优化复杂度，而不是浪费精力在节省一个多余加法这种无关紧要不关乎大局的细节上。

建议 5.7.1.32：尽可能局部地声明变量

摘要：变量将引入状态，状态的存在时间越短越好，最好只作用于用到它的作用域内。

避免污染上下文。

常量不引入状态，不适用于本条。

C++中鼓励即用即声明，有了足够的数据初始化时才声明是一个好选择。

将循环内的局部变量提出循环属于特例，可以自行判别该如何选择。

建议 5.7.1.33：总是初始化变量

摘要：总是再定义变量定义时初始化，避免使用未初始化的变量导致的错误。

一般而言安全性总是优于不必要的性能考虑。

建议 5.7.1.34：积极使用 const

摘要：应尽量使用常量，会带来编译期类型检查。

当需要合法在 const 函数中修改变量时，声明为变量为 mutable（应该用在这种修改不影响对象可观察状态的情况下，比如缓存数据：不影响正确性，只提供更快的性能，数据本身并没有改变）。

const 具有传染性。

不要强制转换 const，这通常意味着设计哪里出现了问题。

应避免将值传递的参数设置为 const，避免在参数中使用顶层 const：在函数声明层面它会被忽略，但是语义约束还在。

5.7.2 规则

规则 5.7.2.1：禁用 C++异常机制

严禁使用 C++的异常机制，所有的错误都应该通过错误值在函数之间传递并做相应的判断, 而不应该通过异常机制进行错误处理。 编码人员必须完全掌控整个编码过程，建立攻击者思维，增强安全编码意识，主动把握有可能出错的环节。而使用 C++异常机制进行错误处理，会削弱编码人员的安全意识。异常机制会打乱程序的正常执行流程，使程序结构更加复杂，原先申请的资源可能会得不到有效清理。异常机制导致代码的复用性降低，使用了异常机制的代码，不能直接给不使用异常机制的代码复用。 异常机制在实现上依赖于编译器、操作系统、处理器，使用异常机制，导致程序执行性能降低。 在二进制层面，程序被加载后，异常处理函数增加了程序的被攻击面，攻击者可以通过覆盖异常处理函数地址，达到攻击的效果。 例外： 在接管 C++语言本身抛出的异常（例如 new 失败、STL）、第三方库（例如 IDL）抛出的异常时，可以使用异常机制，例如：

```
int len = ...;

char *p = NULL;

try

{p = new char[len];

}

catch (bad_alloc) {
```

```
    ...
 abort();
}
```

规则 5.7.2.2：严禁在构造函数中创建线程

构造函数内仅作成员变量的初始化工作，其他的操作通过成员函数完成。

规则 5.7.2.3：禁用 pthread_exit、ExitThread 函数

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。

规则 5.7.2.4：调用格式化函数时，禁止 format 参数由外部可控

调用格式化函数时，如果 format 参数由外部可控，会造成字符串格式化漏洞。 这些格式化函数有：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()。
错误示例:

```
char *msg = GetMsg();
...
printf(msg);
```

推荐做法：

```
char *msg = GetMsg();
...
printf("%s\n", msg);
```

相关指南：

CERT.FIO47-C. Use valid format strings

MITRE.CWE-134: Use of Externally-Controlled Format String

规则 5.7.2.5：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

由于整数在运算过程中可能溢出，当运算结果赋值给比他更大类型，或者和比他更大类型进行比较时，会导致实际结果与预期结果不符。 请观察以下二个代码及其输出：

```
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
```

规则 5.7.2.6：禁止对有符号整数进行位操作符运算

位操作符（～、>>、<<、&、^、|)应该只用于无符号整型操作数。 错误示例：

```
int data = ReadByte();
int a = data >> 24;
```

推荐做法：(为简化示例代码，此处假设 ReadByte 函数实际不存在返回值小于 0 的情况)

```
unsigned int data = (unsigned int)ReadByte();
unsigned int a = data >> 24;
```

在 C 语言中，如果在未知的有符号上执行位操作，很可能会导致缓冲区溢出，从而在某些情况下导致攻击者执行任意代码，同时，还可能会出现出乎意料的行为或编译器定义的行为。

代码如下：

```
#include<stdio.h>
int main()
{
      int y=0x80000000;
       printf("%d\n",y>>24);//以十进制有符号形式输出。
       printf("%u\n",y>>24);//以十进制无符号形式输出。
      return 0;
}
```

输出为：

```
-128

4294967168
```

由于 int 类型的最高位是符号位，剩下的 31 位才用来存储数值，y>>24 的结果应该是 0xffffff80(负数右移，左补 1)，当我们以无符号整型输出时，为正数的 0xffffff80，以有符号整型输出时，应减一再取反，结果为-1289。

然而，在右移运算中，空出的位用 0 还是符号位进行填充是由于具体的 C 语言编译器实现来决定。在通常情况下，如果要进行移位的操作数是无符号类型的，那么空出的位将用 0 进行填充；如果要进行移位的操作数是有符号类型的，则 C 语言编译器实现既可选择 0 来进行填充，也可选择符号位进行填充。

因此，如果很关心一个右移运算中的空位，那么可以使用 unsigned 修饰符来声明变量，这样空位都会被设置为 0。同时，如果一个程序采用了有符号数的右移位操作，那么它就是不可移植的 10。

规则 5.7.2.7：禁止整数与指针间的互相转化

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。

错误示例：

```
char *ptr = ...;

unsigned int number = (unsigned int)ptr;
```

推荐做法：

```
char *ptr = ...;

uintptr_t number = (uintptr_t)ptr;
```

规则 5.7.2.8：禁止对指针进行逻辑或位运算（&&、||、!、～、>>、<<、&、^、|）

对指针进行逻辑运算，会导致指针的性质改变，可能产生内存非法访问的问题。 下面是错误的用法：

```
BOOL DealName(const char *nameA, const char *nameB)
```

```
{
    ...
    if (nameA)
    ...
    if (!nameB)
    ...
}
```

下面是正确的用法：

```
BOOL dealName(const char *nameA, const char *nameB)
{
    ...
    if (nameA != NULL)
    ...
    if (nameB == NULL)
    ...
}
```

例外： 为检查地址对齐而对地址指针进行的位运算可以作为例外。

规则 5.7.2.9：禁止引用未初始化的内存

malloc、new 分配出来的内存没有被初始化为 0，要确保内存被引用前是被初始化的。 以下代码使用 malloc 申请内存，在使用前没有初始化：

```
int *CalcMetrixColomn( int **metrix ,int *param, size_t size )
{
```

```
    int *result = NULL;
    ...
    size_t bufSize = size * sizeof(int);
    ...
    result = (int *)malloc(bufSize);
    ...
    result[0] += metrix[0][0] * param[0];
    ...
    return result;
}
```

以下代码使用 memset_s()对分配出来的内存清零。

```
int *CalcMetrixColomn(int **metrix ,int *param, size_t size)
{
    int *result = NULL;
    ...
    size_t bufSize = size * sizeof(int);
    ...
    result = (int *)malloc(bufSize);
    ...
    int ret = memset_s(result, bufSize, 0, bufSize); //【修改】确保内存被初始化后才被引用
    ...
    result[0] += metrix[0][0] * param[0];
    ...
```

```
    return result;
}
```

规则 5.7.2.10：内存释放之后立即赋予新值

悬挂指针可能会导致双重释放（double-free）以及访问已释放内存的危险。消除悬挂指针以及消除众多与内存相关危险的一个最为有效地方法就是当指针使用完后将其置新值。 如果一个指针释放后能够马上离开作用域，因为它已经不能被再次访问，因此可以无需对其赋予新值。

示例：

```
char *message = NULL;
    ...
    message = (char *)malloc(len);
    ...
    if (...) {
        free(message); //在这个分支内对内存进行了释放
        message = NULL; //释放后将指针赋值为 NULL
    }
    ...
    if (message != NULL) {
        free(message);
        message = NULL;
}
```

规则 5.7.2.11：禁止使用 alloca()函数申请栈上内存

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 或 new，从堆中动态分配内存。

规则 5.7.2.12：创建文件时必须显式指定合适的文件访问权限

创建文件时，如果不显式指定合适访问权限，可能会让未经授权的用户访问该文件。 下列代码没有显式配置文件的访问权限。

```
int fd = open(fileName, O_CREAT | O_WRONLY); //【错误】缺少访问权限设置
```

推荐做法：

```
int fd = open(fileName, O_CREAT | O_WRONLY, S_IRUSR|S_IWUSR);
```

规则 5.7.2.13：严禁使用 string 类存储敏感信息

string 类是 C++内部定义的字符串管理类，如果口令等敏感信息通过 string 进行操作，在程序运行过程中，敏感信息可能会散落到内存的各个地方，并且无法清 0。 以下代码，Foo 函数中获取密码，保存到 string 变量 password 中，随后传递给 VerifyPassword 函数，在这个过程中，password 实际上在内存中出现了 2 份。

```
int VerifyPassword(string password)
{
    //...
}
int Foo()
{
    string password = GetPassword();
    VerifyPassword(password);
    ...
```

```
}
```

应该使用 char 或 unsigned char 保存敏感信息，如下代码：

```
int VerifyPassword(const char *password)
{
    //...
}
int Foo()
{
    char password[MAX_PASSWORD] = {0};
    GetPassword(password, sizeof(password));
    VerifyPassword(password);
...
}
```

规则 5.7.2.14：注意隐式符号转换

两个无符号数相减为负数时，结果应当为一个很大的无符号数，但是小于 int 的无符号数在运算时可能会有预期外的隐式符号转换。

```
// 1
unsigned char a = 1;unsigned char b = 2;
if (a - b < 0)  // a - b = -1 (signed int) a = 6;
else a = 8;
// 2unsigned char a = 1;unsigned short b = 2;
```

```
    if (a - b < 0)  // a - b = -1 (signed int)  a = 6;else a = 8;
```

上述结果均为 a=6

```
    // 3unsigned int a = 1;unsigned short b = 2;

    if (a - b < 0)  // a - b = 0xffffffff (unsigned int)a = 6;elsea = 8;

    // 4unsigned int a = 1;unsigned int b = 2;

    if (a - b < 0) // a - b = 0xffffffff (unsigned int)a = 6;elsea = 8;
```

上述结果均为 a=8

如果预期为 8，则错误代码：

```
    // Bad

    unsigned short a = 1;unsigned short b = 2;

    if (a - b < 0)  // a - b = -1 (signed int) a = 6;else a = 8;
```

正确代码：

```
    // Good

    unsigned short a = 1;unsigned short b = 2;

    if ((unsigned int)a - (unsigned int)b < 0)  // a - b = 0xffff (unsigned short) a = 6; else
a = 8;
```

规则 5.7.2.15：注意八进制问题

代码对齐时应当使用空格或者编辑器自带的对齐功能，谨慎在数字前使用 0 来对齐代码，以免不当将某些内容转换为八进制。

例如，如果预期为 20 字节长度的缓冲区，则下列代码存在错误。buf2 为 020（OCT）长度，实际只有 16（DEC）长度，在 memcpy 后越界：

```
    // Bad
```

```
char buf1[1024] = {0};char buf2[0020] = {0};
memcpy(buf2, somebuf, 19);
```

应当在使用 8 进制时明确注明这是八进制。

```
// Goodint access_mask = 0777;  // oct, rwxrwxrwx
```

规则 5.7.2.16：返回栈上变量的地址

函数不可以返回栈上的变量的地址，其内容在函数返回后就会失效。

```
// Bad
char* Foo(char* sz, int len){
   char a[300] = {0};
   if (len > 100) {
      memcpy(a, sz, 100);
   }
   a[len] = '\0';
   return a;  // WRONG
}
```

而应当使用堆来传递非简单类型变量。

```
// Good
char* Foo(char* sz, int len) {
      char* a = new char[300];
      if (len > 100) {
            memcpy(a, sz, 100);
      }
```

```
        a[len] = '\0';
        return a;    // OK
}
```

对于 C++ 程序来说，强烈建议返回 string、vector 等类型，会让代码更加简单和安全。

规则 5.7.2.17：有逻辑联系的数组必须仔细检查

例如下列程序将字符串转换为 week day，但是两个数组并不一样长，导致程序可能会越界读一个 int。

```
// Bad
int nWeekdays[] = {1, 2, 3, 4, 5, 6};
const char* sWeekdays[] = {"Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"};
for (int x = 0; x < ARRAY_SIZE(sWeekdays); x++) {
   if (strcmp(sWeekdays[x], input) == 0)
       return nWeekdays[x];
}
```

应当确保有关联的 nWeekdays 和 sWeekdays 数据统一。

```
// Good
const int nWeekdays[] = {1, 2, 3, 4, 5, 6, 7};
const char* sWeekdays[] = {"Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"};
assert(ARRAY_SIZE(nWeekdays) == ARRAY_SIZE(sWeekdays));
for (int x = 0; x < ARRAY_SIZE(sWeekdays); x++) {
   if (strcmp(sWeekdays[x], input) == 0) {
       return nWeekdays[x];
   }
```

```
}
```

规则 5.7.2.18：左右一致的重复判断/永远为真或假的判断（通常代表错误）

这通常是由于自动完成或例如 Visual Assistant X 之类的补全插件导致的问题。

```
// Bad
if (foo1.bar == foo1.bar) {
    …
}
```

可能是：

```
// Good
if (foo1.bar == foo2.bar) {
    …
}
```

规则 5.7.2.19：不得使用栈上未初始化的变量

在栈上声明的变量要注意是否在使用它之前已经初始化了

```
// Bad
void Foo() {
    int foo;
    if (Bar()) {
        foo = 1;
    }
    Foobar(foo); // foo 可能没有初始化
```

```
}
```

最好在声明的时候就立刻初始化变量，或者确保每个分支都初始化它。开启相应的编译器警告（GCC 中为 -Wuninitialized），并把设置为错误级别，可以在编译阶段发现这类错误。

```
// Goodvoid Foo() {
  int foo = 0;
  if (Bar()) {
    foo = 1;
  }
  Foobar(foo);
}
```

规则 5.7.2.20：变量应确保线程安全性

当一个变量可能被多个线程使用时，应当使用原子操作或加锁操作。

```
// Bad
char  g_somechar;
void foo_thread1() {
  g_somechar += 3;
}
void foo_thread2() {
  g_somechar += 1;
}
```

对于可以使用原子操作的，应当使用一些可以确保内存安全的操作，如：

```
// Good
char g_somechar;
```

```
void foo_thread1() {
    __sync_fetch_and_add(&g_somechar, 3);
}
void foo_thread2() {
    __sync_fetch_and_add(&g_somechar, 1);
}
```

对于 C 代码，C11 后推荐使用 atomic 标准库。 对于 C++代码，C++11 后，推荐使用 std::atomic。

规则 5.7.2.21：注意 signal handler 导致的条件竞争

竞争条件经常出现在信号处理程序中，因为信号处理程序支持异步操作。攻击者能够利用信号处理程序争用条件导致软件状态损坏，从而可能导致拒绝服务甚至代码执行。

当信号处理程序中发生不可重入函数或状态敏感操作时，就会出现这些问题。因为信号处理程序中随时可以被调用。比如，当在信号处理程序中调用 free 时，通常会出现另一个信号争用条件，从而导致双重释放。即使给定指针在释放后设置为 NULL，在释放内存和将指针设置为 NULL 之间仍然存在竞争的可能。

为多个信号设置了相同的信号处理程序，这尤其有问题——因为这意味着信号处理程序本身可能会重新进入。例如，malloc()和 free()是不可重入的，因为它们可能使用全局或静态数据结构来管理内存，并且它们被 syslog()等看似无害的函数间接使用；这些函数可能会导致内存损坏和代码执行。

```
// Bad
char *log_message;
void Handler(int signum) {
    syslog(LOG_NOTICE, "%s\n", log_m_essage);
    free(log_message);
    sleep(10);
    exit(0);
}
```

```
int main (int argc, char* argv[]) {
    log_message = strdup(argv[1]);
    signal(SIGHUP, Handler);
    signal(SIGTERM, Handler);
    sleep(10);
}
```

可以借由下列操作规避问题：

避免在多个处理函数中共享某些变量。

在信号处理程序中使用同步操作。

屏蔽不相关的信号，从而提供原子性。

避免在信号处理函数中调用不满足异步信号安全的函数。

规则 5.7.2.22：注意 Time-of-check Time-of-use (TOCTOU) 条件竞争

TOCTOU： 软件在使用某个资源之前检查该资源的状态，但是该资源的状态可以在检查和使用之间更改，从而使检查结果无效。当资源处于这种意外状态时，这可能会导致软件执行错误操作。

当攻击者可以影响检查和使用之间的资源状态时，此问题可能与安全相关。这可能发生在共享资源(如文件、内存，甚至多线程程序中的变量)上。在编程时需要注意避免出现 TOCTOU 问题。

例如，下面的例子中，该文件可能已经在检查和 lstat 之间进行了更新，特别是因为 printf 有延迟。

```
struct stat *st;
lstat("...", st);
printf("foo");
if (st->st_mtimespec == ...) {
    printf("Now updating things\n");
    UpdateThings();
}
```

TOCTOU 难以修复，但是有以下缓解方案：

限制对来自多个进程的文件的交叉操作。

如果必须在多个进程或线程之间共享对资源的访问，那么请尝试限制”检查“（CHECK）和”使用“（USE）资源之间的时间量，使他们相距尽量不要太远。这不会从根本上解决问题，但可能会使攻击更难成功。

在 Use 调用之后重新检查资源，以验证是否正确执行了操作。

确保一些环境锁定机制能够被用来有效保护资源。但要确保锁定是检查之前进行的，而不是在检查之后进行的，以便检查时的资源与使用时的资源相同。

规则 5.7.2.23：避免路径穿越问题

在进行文件操作时，需要判断外部传入的文件名是否合法，如果文件名中包含 ../ 等特殊字符，则会造成路径穿越，导致任意文件的读写。

错误：

```
void Foo() {
    char file_path[PATH_MAX] = "/home/user/code/";
    // 如果传入的文件名包含../可导致路径穿越
    // 例如"../file.txt"，则可以读取到上层目录的 file.txt 文件
    char name[20] = "../file.txt";
    memcpy(file_path + strlen(file_path), name, sizeof(name));
    int fd = open(file_path, O_RDONLY);
    if (fd != -1) {
        char data[100] = {0};
        int num = 0;
        memset(data, 0, sizeof(data));
        num = read(fd, data, sizeof(data));
        if (num > 0) {
```

```
            write(STDOUT_FILENO, data, num);
        }
        close(fd);
    }
}
```

正确：

```
void Foo() {
    char file_path[PATH_MAX] = "/home/user/code/";
    char name[20] = "../file.txt";
    // 判断传入的文件名是否非法，例如"../file.txt"中包含非法字符../，直接返回
    if (strstr(name, "..") != NULL){
        // 包含非法字符
        return;
    }
    memcpy(file_path + strlen(file_path), name, sizeof(name));
    int fd = open(file_path, O_RDONLY);
    if (fd != -1) {
        char data[100] = {0};
        int num = 0;
        memset(data, 0, sizeof(data));
        num = read(fd, data, sizeof(data));
        if (num > 0) {
            write(STDOUT_FILENO, data, num);
```

```
        }
        close(fd);
      }
  }
```

规则 5.7.2.24：避免相对路径导致的安全问题（DLL、EXE 劫持等问题）

在程序中，使用相对路径可能导致一些安全风险，例如 DLL、EXE 劫持等问题。

例如以下代码，可能存在劫持问题：

```
int Foo() {
   // 传入的是 dll 文件名，如果当前目录下被写入了恶意的同名 dll，则可能导致 dll 劫持
   HINSTANCE hinst = ::LoadLibrary("dll_nolib.dll");
   if (hinst != NULL) {
      cout<<"dll loaded!" << endl;
   }
   return 0;
}
```

针对 DLL 劫持的安全编码的规范：

1）调用 LoadLibrary，LoadLibraryEx，CreateProcess，ShellExecute 等进行模块加载的函数时，指明模块的完整（全）路径，禁止使用相对路径，这样就可避免从其它目录加载 DLL。 2）在应用程序的开头调用 SetDllDirectory(TEXT("")); 从而将当前目录从 DLL 的搜索列表中删除。结合 SetDefaultDllDirectories，AddDllDirectory，RemoveDllDirectory 这几个 API 配合使用，可以有效的规避 DLL 劫持问题。这些 API 只能在打了 KB2533623 补丁的 Windows7，2008 上使用。

规则 5.7.2.25：文件权限控制

在创建文件时，需要根据文件的敏感级别设置不同的访问权限，以防止敏感数据被其他恶意程序读

取或写入。

错误：

```
int Foo() {
   // 不要设置为 777 权限，以防止被其他恶意程序操作
   if (creat("file.txt", 0777) < 0) {
      printf("文件创建失败！\n");
   } else {
      printf("文件创建成功！\n");
   }
   return 0;
}
```

规则 5.7.2.26：防止各种越界写（向前/向后）

```
错误 1：
int a[5];
a[5] = 0;
错误 2：
int a[5];int b = user_controlled_value;
a[b] = 3;
```

规则 5.7.2.27：防止任意地址写

任意地址写会导致严重的安全隐患，可能导致代码执行。因此，在编码时必须校验写入的地址。

```
错误：
```

```
void Write(MyStruct dst_struct) {
    char payload[10] = { 0 };
    memcpy(dst_struct.buf, payload, sizeof(payload));
}
int main() {
    MyStruct dst_stuct;
    dst_stuct.buf = (char*)user_controlled_value;
    Write(dst_stuct);
    return 0;
}
```

规则 5.7.2.28：必须防止 Off-By-One

在进行计算或者操作时，如果使用的最大值或最小值不正确，使得该值比正确值多 1 或少 1，可能导致安全风险。

```
错误：
char firstname[20];char lastname[20];char fullname[40];
fullname[0] = '\0';
strncat(fullname, firstname, 20);// 第二次调用 strncat()可能会追加另外 20 个字符。如果这 20 个字符没有终止空字符，则存在安全问题 strncat(fullname, lastname, 20);
正确：
char firstname[20];char lastname[20];char fullname[40];
fullname[0] = '\0';
```

```
// 当使用像 strncat()函数时，必须在缓冲区的末尾为终止空字符留下一个空字节，避免 off-by-onestrncat(fullname, firstname, sizeof(fullname) - strlen(fullname) - 1);strncat(fullname, lastname, sizeof(fullname) - strlen(fullname) - 1);
```

对于 C++ 代码，再次强烈建议使用 string、vector 等组件代替原始指针和数组操作。

规则 5.7.2.29：检查除以零异常

在进行除法运算时，需要判断被除数是否为零，以防导致程序不符合预期或者崩溃。

```
错误：

int divide(int x, int y) {
    return x / y;
}

正确：

int divide(int x, int y) {
    if (y == 0) {
        throw DivideByZero;
    }
    return x / y;
}
```

规则 5.7.2.30：防止数字类型的错误强转

在有符号和无符号数字参与的运算中，需要注意类型强转可能导致的逻辑错误，建议指定参与计算时数字的类型或者统一类型参与计算。

```
错误例子

int Foo() {
```

```
    int len = 1;
    unsigned int size = 9;
    // 1 < 9 - 10？由于运算中无符号和有符号混用，导致计算结果以无符号计算
    if (len < size - 10) {
        printf("Bad\n");
    } else {
        printf("Good\n");
    }
}
```

正确例子

```
void Foo() {
    // 统一两者计算类型为有符号
    int len = 1;
    int size = 9;
    if (len < size - 10) {
        printf("Bad\n");
    } else {
        printf("Good\n");
    }
}
```

规则 5.7.2.31：比较数据大小时加上最小/最大值的校验

在进行数据大小比较时，要合理地校验数据的区间范围，建议根据数字类型，对其进行最大和最小值的判断，以防止非预期错误。

错误：

```
void Foo(int index) {
  int a[30] = {0};
  // 此处 index 是 int 型，只考虑了 index 小于数组大小，但是并未判断是否大于等于 0
  if (index < 30) {
    // 如果 index 为负数，则越界
    a[index] = 1;
  }
}
```

正确：

```
void Foo(int index) {
  int a[30] = {0};
  // 判断 index 的最大最小值
  if (index >= 0 && index < 30) {
    a[index] = 1;
  }
}
```

规则 5.7.2.32：检查在 pointer 上使用 sizeof

除了测试当前指针长度，否则一般不会在 pointer 上使用 sizeof。

正确：

```
size_t pointer_length = sizeof(void*);
```

可能错误：

```
size_t structure_length = sizeof(Foo*);
```

可能是：

```
size_t structure_length = sizeof(Foo);
```

规则 5.7.2.33：检查直接将数组和 0 比较的代码

错误：

```
int a[3];
...;
if (a > 0)
   ...;
```

该判断永远为真，等价于:

```
int a[3];
...;
if (&a[0])
   ...;
```

可能是：

```
int a[3];
...;
if(a[0] > 0)
   ...;
```

开启足够的编译器警告（GCC 中为 -Waddress，并已包含在 -Wall 中），并设置为错误，可以在编译期间发现该问题。

规则 5.7.2.34：不应当向指针赋予写死的地址

特殊情况需要特殊对待（比如开发硬件固件时可能需要写死）
但是如果是系统驱动开发之类的，写死可能会导致后续的问题。

规则 5.7.2.35：释放完后置空指针

在对指针进行释放后，需要将该指针设置为 NULL，以防止后续 free 指针的误用，导致 UAF 等其他内存破坏问题。尤其是在结构体、类里面存储的原始指针。

错误：

```
void foo() {
    char* p = (char*)malloc(100);
    memcpy(p, "hello", 6);
    printf("%s\n", p);
    free(p); // 此时 p 所指向的内存已被释放，但是 p 所指的地址仍然不变
    // 未设置为 NULL，可能导致 UAF 等内存错误

    if (p != NULL) {  // 没有起到防错作用
        printf("%s\n", p); // 错误使用已经释放的内存
    }
}
```

正确：

```
void foo() {
    char* p = (char*)malloc(100);
```

```
    memcpy(p, "hello", 6);
    // 此时 p 所指向的内存已被释放，但是 p 所指的地址仍然不变
    printf("%s\n", p);
    free(p);
    //释放后将指针赋值为空 p = NULL;
    if (p != NULL)  { // 没有起到防错作用
      printf("%s\n", p); // 错误使用已经释放的内存
    }
}
```

对于 C++ 代码，使用 string、vector、智能指针等代替原始内存管理机制，可以大量减少这类错误。

规则 5.7.2.36：防止错误的类型转换（type confusion）

在对指针、对象或变量进行操作时，需要能够正确判断所操作对象的原始类型。如果使用了与原始类型不兼容的类型进行访问，则存在安全隐患。

```
错误：
const int NAME_TYPE = 1;const int ID_TYPE = 2;
// 该类型根据 msg_type 进行区分，如果在对 MessageBuffer 进行操作时没有判断目标对象，则存在类型混淆 struct MessageBuffer {
    int msg_type;
    union {
      const char *name;
      int name_id;
    };
```

```
};
void Foo() {
    struct MessageBuffer buf;
    const char* default_message = "Hello World";
    // 设置该消息类型为 NAME_TYPE，因此 buf 预期的类型为 msg_type + name
    buf.msg_type = NAME_TYPE;
    buf.name = default_message;
    printf("Pointer of buf.name is %p\n", buf.name);

    // 没有判断目标消息类型是否为 ID_TYPE，直接修改 nameID，导致类型混淆
    buf.name_id = user_controlled_value;

    if (buf.msg_type == NAME_TYPE) {
        printf("Pointer of buf.name is now %p\n", buf.name);
        // 以 NAME_TYPE 作为类型操作，可能导致非法内存读写
        printf("Message: %s\n", buf.name);
    } else {
        printf("Message: Use ID %d\n", buf.name_id);
    }
}
```

正确（判断操作的目标是否是预期类型）：

```
void Foo() {
    struct MessageBuffer buf;
```

```
    const char* default_message = "Hello World";
    // 设置该消息类型为 NAME_TYPE，因此 buf 预期的类型为 msg_type + name
    buf.msg_type = NAME_TYPE;
    buf.name = default_msessage;
    printf("Pointer of buf.name is %p\n", buf.name);
    // 判断目标消息类型是否为 ID_TYPE，不是预期类型则做对应操作
    if (buf.msg_type == ID_TYPE)
        buf.name_id = user_controlled_value;
    if (buf.msg_type == NAME_TYPE) {
        printf("Pointer of buf.name is now %p\n", buf.name);
        printf("Message: %s\n", buf.name);
    } else {
        printf("Message: Use ID %d\n", buf.name_id);
    }
}
```

规则 5.7.2.37：智能指针使用安全

在使用智能指针时，防止其和原始指针的混用，否则可能导致对象生命周期问题，例如 UAF 等安全风险。

错误例子：

```
class Foo {
  public:
    explicit Foo(int num) { data_ = num; };
```

```
    void Function() { printf("Obj is %p, data = %d\n", this, data_); };
  private:
    int data_;
};

std::unique_ptr<Foo> fool_u_ptr = nullptr;
Foo* pfool_raw_ptr = nullptr;
void Risk() {
    fool_u_ptr = make_unique<Foo>(1);

    // 从独占智能指针中获取原始指针,<Foo>(1)
    pfool_raw_ptr = fool_u_ptr.get();
    // 调用<Foo>(1)的函数
    pfool_raw_ptr->Function();

    // 独占智能指针重新赋值后会释放内存
    fool_u_ptr = make_unique<Foo>(2);
    // 通过原始指针操作会导致 UAF，pfool_raw_ptr 指向的对象已经释放
    pfool_raw_ptr->Function();
}
// 输出：// Obj is 0000027943087B80, data = 1// Obj is 0000027943087B80, data = -572662307
```

正确，通过智能指针操作:

```
void Safe() {
    fool_u_ptr = make_unique<Foo>(1);
    // 调用<Foo>(1)的函数
    fool_u_ptr->Function();

    fool_u_ptr = make_unique<Foo>(2);
    // 调用<Foo>(2)的函数
    fool_u_ptr->Function();
}
// 输出：// Obj is 000002C7BB550830, data = 1// Obj is 000002C7BB557AF0, data = 2
```

规则 5.7.2.38：不可失去对已分配资源的控制

对于动态分配的资源，其地址、句柄或描述符等标志性信息不可被遗失，否则资源无法被访问也无法被回收，这种问题称为“资源泄漏”，会导致资源耗尽或死锁等问题，使程序无法正常运行。

在资源被回收之前，记录其标志性信息的变量如果：

均被重新赋值

生命周期均已结束

所在线程均被终止

相关资源便失去了控制，无法再通过正常手段访问相关资源。

示例：

```
int fd;
fd = open("a", O_RDONLY);     // Open a file descriptor
read(fd, buf1, 100);
fd = open("b", O_RDONLY);     // Non-compliant, the previous descriptor is lost
read(fd, buf2, 100);
```

例中变量 fd 记录文件资源描述符，在回收资源之前对其重新赋值会导致资源泄漏。

规则 5.7.2.39：不可失去对已分配内存的控制

动态分配的内存地址不可被遗失，否则相关内存无法被访问也无法被回收，这种问题称为“内存泄漏（memory leak）”，会导致可用内存被耗尽，使程序无法正常运行。

程序需要保证内存分配与回收之间的流程可达，且不可被异常中断，相关线程也不可在中途停止。

本规则是 ID_resourceLeak 的特化。

示例：

```
void foo(size_t n) {
    void* p = malloc(n);
    if (cond) {
        return;   // Non-compliant, ‘p’ is lost
    }
    ....
    free(p);
}
```

例中局部变量 p 记录已分配的内存地址，释放前在某种情况下函数返回，之后便再也无法访问到这块内存了，导致内存泄露。

又如：

```
void bar(size_t n) {
    void* p = malloc(n);
    if (n < 100) {
        p = realloc(p, 100);   // Non-compliant, the original value of ‘p’ is lost
    }
```

```
    ....
}
```

例中 realloc 函数分配失败会返回 NULL，p 未经释放而被重新赋值，导致内存泄露。

规则 5.7.2.40：资源源应接受对象化管理

资源接受对象化管理，免去繁琐易错的手工分配回收过程，是 C++ 程序设计的重要方法。

将资源分配的结果直接在程序中传递是非常不安全的，极易产生泄漏或死锁等问题。动态申请的资源如果只用普通变量引用，不受对象的构造或析构机制控制，则称为“无主”资源，在 C++ 程序设计中应当避免。

应尽量使用标准库提供的容器或智能指针，避免显式使用资源管理接口。本规则集合示例中的 new/delete、lock/unlock 意在代指一般的资源操作，仅作示例，在实际代码中应尽量避免。

示例：

```
void foo(size_t size) {
    int* p = new int[size];      // Bad, ownerless
    ....                                  // If any exception is thrown, or a wrong jump, leak
    delete[] p;
}


struct X {
    int* p;
};


void bar() {X x;
    x.p = new int[123];      // Bad, ‘X’ has no destructor, ‘x’ is not an owner
    ....
```

```
}


class Y {
    int* p;
public:
    Y(size_t n): p(new int[n]) {}
   ~Y() { delete[] p; }
};
void baz() {
    Y y(123);     // Good, ‘y’ is the owner of the resource
    ....
}
```

例中 foo 和 bar 函数的资源管理方式是不符合 C++ 理念的，baz 函数中的 y 对象负责资源的分配与回收，称 y 对象具有资源的所有权，相关资源的生命周期与 y 的生命周期一致，有效避免了资源泄漏或错误回收等问题。

资源的所有权可以发生转移，但应保证转移前后均有对象负责管理资源，并且在转移过程中不会产生异常。进一步理解对象化管理方法，可参见“RAII（Resource Acquisition Is Initialization）”等机制。

与资源相关的系统接口不应直接被业务代码引用，如：

```
void foo(const TCHAR* path) {
    HANDLE h;
    WIN32_FIND_DATA ffd;

    h = FindFirstFile(path, &ffd);   // Bad, ownerless
    ....
```

```
    CloseHandle(h);    // Is it right?
}
```

例中 Windows API FindFirstFile 返回资源句柄，是“无主”资源，很可能被后续代码误用或遗忘。应进行合理封装：

```
class MY_FIND_DATA
{
    struct HANDLE_DELETER
    {
        using pointer = HANDLE;
        void operator()(pointer p) { FindClose(p); }
    };
    WIN32_FIND_DATA ffd;
    unique_ptr<HANDLE, HANDLE_DELETER> uptr;

public:
    MY_FIND_DATA(const TCHAR* path): uptr(FindFirstFile(path, &ffd)) {}
    ....
    HANDLE handle() { return uptr.get(); }
};
```

本例将 FindFirstFile 及其相关数据封装成一个类，由 unique_ptr 对象保存 FindFirstFile 的结果，FindClose 是资源的回收方法，将其作为 unique_ptr 对象的组成部分，使资源可以被自动回收。

规则 5.7.2.41：资源源的分配与回收方法应成对提供

资源的分配和回收方法应在同一库或主程序等可执行模块、类等逻辑模块中提供。

如果一个模块分配的资源需要另一个模块回收，会打破模块之间的独立性，增加维护成本，而且 so、dll、exe 等可执行模块一般都有独立的堆栈，跨模块的分配与回收往往会造成严重错误。

示例：

```
// In a.dll
int* foo() {
    return (int*)malloc(1024);
}

// In b.dll
void bar() {
    int* p = foo();
    ....
    free(p);    // Non-compliant, crash
}
```

例中 a.dll 分配的内存由 b.dll 释放，相当于混淆了不同堆栈中的数据，程序一般会崩溃。

应改为：

```
// In a.dll
int* foo_alloc() {
    return (int*)malloc(1024);
}

void foo_dealloc(int* p) {
    free(p);
}
```

```
// In b.dll
void bar() {
    int* p = foo_alloc();
    ....
    foo_dealloc(p);     // Compliant
}
```

修正后 a.dll 成对提供分配回收函数，b.dll 配套使用这些函数，避免了冲突。

类等逻辑模块提供了分配方法，也应提供回收方法，如重载了 new 运算符，也应重载相应的 delete 运算符：

```
class A {
    void* operator new(size_t);     // Non-compliant, missing ‘operator delete’
};

class B {
    void operator delete(void*);     // Non-compliant, missing ‘operator new’
};

class C {
    void* operator new(size_t);     // Compliant
    void operator delete(void*);     // Compliant
};
```

placement-new 与 placement-delete 也应成对提供：

```
class D {
    void* operator new(size_t, bool);      // Non-compliant


    void* operator new(size_t, int, int);      // Compliant
    void operator delete(void*, int, int);      // Compliant
};
```

规则 5.7.2.42：资源的分配与回收方法应配套使用

使用了某种分配方法，就应使用与其配套的回收方法，否则会引发严重错误。

示例：

```
void foo() {
    T* p = new T;
    ....
    free(p);      // Non-compliant, use ‘delete’ instead
}


void bar(size_t n) {
    char* p = (char*)malloc(n);
    ....
    delete[] p;      // Non-compliant, use ‘free’ instead
}
```

不同的分配回收方法属于不同的资源管理体系，用 new 分配的资源应使用 delete 回收，malloc 分配的应使用 free 回收。

规则 5.7.2.43：不应在模块之间传递容器类对象

在库或主程序等可执行模块之间传递容器类对象会造成分配回收方面的冲突。

与资源管理相关的对象，如流、字符串、智能指针以及自定义对象均不应在模块间传递。

不同的可执行模块往往具有独立的资源管理机制，跨模块的分配与回收会造成严重错误，而且不同的模块可能由不同的编译器生成，对同一对象的实现也可能存在冲突。

示例：

```
// In a.dll
void foo(vector<int>& v) {v.reserve(100);
}

// In b.exe
int main() {
    vector<int> v {     // Allocation in b.exe    1, 2, 3
    };
    foo(v);     // Non-compliant, reallocation in a.dll, crash
}
```

例中容器 v 的初始内存由 b.exe 分配，b.exe 与 a.dll 具有独立的堆栈，由于模板库的内联实现，reserve 函数会调用 a.dll 的内存管理函数重新分配 b.exe 中的内存，造成严重错误。

规则 5.7.2.44：不应在模块之间传递非标准布局类型的对象

非标准布局类型的运行时特性依赖编译器的具体实现，在不同编译器生成的模块间传递这种类型的对象会导致运行时错误。

“标准布局（standard-layout）”类型的主要特点：

没有虚函数也没有虚基类

所有非静态数据成员均具有相同的访问权限

所有非静态数据成员和位域都在同一个类中声明

不存在相同类型的基类对象

没有非标准布局的基类

没有非标准布局和引用类型的非静态数据成员

除非模块均由同一编译器的同一版本生成，否则不具备上述特点的对象不应在模块之间传递。

示例：

```
// a.dll
class A {
    ....
public:
    virtual void foo();     // Non standard-layout
};
void bar(A&);
// b.exe
int main() {A a;
bar(a);     // Non-compliant
}
```

设例中 a.dll 和 b.exe 由不同的编译器生成，b.exe 中定义的 a 对象被传递给了 a.dll 中定义的接口，由于存在虚函数，不同的编译器对 a 对象的内存布局会有不同的解读，从而造成冲突。

规则 5.7.2.45：对象申请的资源应在析构函数中释放

对象在析构函数中释放自己申请的资源是 C++ 程序设计的重要原则，不可被遗忘，也不应要求用户释放。

示例：

```
class A {
    int* p = nullptr;
```

```
public:
    A(size_t n): p(new int[n]) {
    }

   ~A() {    // Non-compliant, must delete[] p
    }
};
```

例中成员 p 与内存分配有关，但析构函数为空，不符合本规则要求。

规则 5.7.2.46：对象被移动后应重置状态再使用

对象被移动后在逻辑上不再有效，如果没有通过清空数据或重新初始化等方法更新对象的状态，不应再使用该对象。

示例：

```
#include <vector>

using V = std::vector<int>;

void foo(V& a, V& b)
{
    a = std::move(b);      // After moving, the state of ‘b’ is unspecified
    b.push_back(0);         // Non-compliant
}
```

例中容器 b 的数据被移动到容器 a，可能是通过交换的方法实现的，也可能是通过其他方法实现

的，标准容器被移动后的状态在 C++ 标准中是未声明的，程序不应依赖未声明的状态。

应改为：

```
void foo(V& a, V& b)
{       a = std::move(b); b.clear();                // Clear    b.push_back(0);         // Compliant
}
```

规则 5.7.2.47：构造函数抛出异常需避免相关资源泄漏

构造函数抛出异常表示对象构造失败，不会再执行相关析构函数，需要保证已分配的资源被有效回收。

示例：

```
class A {
    int *a, *b;

public:
    A(size_t n):
        a(new int[n]),
        b(new int[n])         // The allocations may fail
    {
        if (sth_wrong) {
            throw E();        // User exceptions
        }
    }
   ~A() {                          // May be invalid
```

```
            delete[] a;
            delete[] b;
        }
    };
```

例中内存分配可能会失败，抛出 bad_alloc 异常，在某种条件下还会抛出自定义的异常，任何一种异常被抛出析构函数就不会被执行，已分配的资源就无法被回收，但已构造完毕的对象还是会正常析构的，所以应采用对象化资源管理方法，使资源可以被自动回收。

可改为：

```
A::A(size_t n) { // Use objects to hold resources
    auto holder_a = make_unique<int[]>(n);
    auto holder_b = make_unique<int[]>(n);
    // Do the tasks that may throw exceptions
    if (sth_wrong) {
        throw E();
    } a = holder_a.release();    b = holder_b.release();
}
```

先用 unique_ptr 对象持有资源，完成可能抛出异常的事务之后，再将资源转移给相关成员，转移的过程不可抛出异常，这种模式可以保证异常安全，如果有异常抛出，资源均可被正常回收。对遵循 C++11 及之后标准的代码，建议用 make_unique 函数代替 new 运算符。

示例代码意在讨论一种通用模式，实际代码可采用更直接的方式：

```
class A {
    vector<int> a, b;    // Or use ‘unique_ptr’

public:
```

```
    A(size_t n): a(n), b(n) {    // Safe and brief
        ....
    }
};
```

保证已分配的资源时刻有对象负责回收是重要的设计原则，可参见 ID_ownerlessResource 的进一步讨论。

注意，“未成功初始化的对象”在 C++ 语言中是不存在的，应避免相关逻辑错误，如：

```
struct T {
    A() { throw CtorException(); }
};
void foo() {
    T* p = nullptr;
    try {p = new T;
    }
    catch (CtorException&) {
        delete p;                    // Logic error, ‘p’ is nullptr
        return;
    }
    ....
    delete p;
}
```

例中 T 类型的对象在构造时抛出异常，而实际上 p 并不会指向一个未能成功初始化的对象，赋值被异常中断，catch 中的 p 仍然是一个空指针，new 表达式中抛出异常会自动回收已分配的内存。

规则 5.7.2.48：资源不可被重复释放

重复释放资源属于逻辑错误，导致标准未定义的行为。

示例：

```
void foo(const char* path) {
    FILE* p = fopen(path, "r");
    if (p) {
        ....
        fclose(p);
    }
    fclose(p);    // Non-compliant
}
```

规则 5.7.2.49：用 delete 释放对象需保证其类型完整

如果用 delete 释放不完整类型的对象，而对象完整类型声明中有 non-trivial 析构函数，会导致标准未定义的行为。

示例：

```
struct T;

void foo(T* p) {
    delete p;            // Non-compliant, undefined behavior
}

struct T {
```

```
    ~T();                   // Non-trivial destructor
};
```

例中 delete 作用于不完整类型的指针 p，析构函数不会正确执行，应保证 T 在 foo 之前定义：

```
struct T {
    ~T();
};

void foo(T* p) {
    delete p;           // Compliant
}
```

规则 5.7.2.50：用 delete 释放对象不可多写中括号

用 new 分配的对象应该用 delete 释放，不可用 delete[] 释放，否则导致标准未定义的行为。

示例：

```
auto* p = new X;    // One object
....
delete[] p;             // Non-compliant, use ‘delete p;’ instead
```

规则 5.7.2.51：用 delete 释放数组不可漏写中括号

用 new 分配的数组应该用 delete[] 释放，不可漏写中括号，否则导致标准未定义的行为。

示例：

```
void foo(int n) {
    auto* p = new X[n];    // n default-constructed X objects
    ....
```

```
    delete p;                    // Non-compliant, use ‘delete[] p;’ instead
}
```

在某些环境中，可能只有第一个对象的析构函数被执行，其他对象的析构函数都没有被执行，如果对象与资源分配有关就会导致资源泄漏。

规则 5.7.2.52：非动态申请的资源不可被释放

释放非动态申请的资源会导致标准未定义的行为。

示例：

```
void foo(size_t n) {
    int* p = (int*)alloca(n);
    ....
    free(p);       // Non-compliant, ‘p’ should not be freed
}

void bar() {
    int i;
    ....
    free(&i);      // Non-compliant, naughty behavior
}
```

释放在栈上分配的空间或者局部对象的地址会造成严重的运行时错误。

规则 5.7.2.53：在一个表达式语句中最多使用一次 new

如果表达式语句多次使用 new，一旦某个构造函数抛出异常就会造成内存泄漏。

示例：

```
fun(
```

```
        shared_ptr<T>(new T),
        shared_ptr<T>(new T)     // Non-compliant, potential memory leak
    );
```

例中 fun 的两个参数均为 new 表达式，实际执行时可以先为两个对象分配内存，再分别执行对象的构造函数，如果某个构造函数抛出异常，已分配的内存就得不到回收了。

保证一次内存分配对应一个构造函数可解决这种问题：

```
auto a(shared_ptr<T>(new T));     // Compliant
auto b(shared_ptr<T>(new T));     // Compliant
fun(a, b);
```

这样即使构造函数抛出异常也会自动回收已分配的内存。

更好的方法是避免显式资源分配：

```
fun(
    make_shared<T>(),
    make_shared<T>()       // Compliant, safe and brief
);
```

用 make_shared、make_unique 等函数代替 new 运算符可有效规避这种问题。

规则 5.7.2.54：流式资源对象不应被复制

FILE 等流式对象不应被复制，如果存在多个副本会造成数据不一致的问题。

示例：

```
FILE f;
FILE* fp = fopen(path, "r"); f = *fp;                    // Non-compliant
memcpy(fp, &f, sizeof(*fp));    // Non-compliant
```

规则 5.7.2.55：避免使用变长数组

使用变长数组（variable length array）可以在栈上动态分配内存，但分配失败时的行为不受程序控制。变长数组由 C99 标准提出，不在 C++ 标准之内，在 C++ 代码中不应使用。

示例：

```
void foo(int n)
{
    int a[n];     // Non-compliant, a variable length array
                  // Undefined behavior if n <= 0
    ....
}
```

例中数组 a 的长度为变量，其内存空间在运行时动态分配，如果长度参数 n 不是合理的正整数会导致未定义的行为。

另外，对于本应兼容的数组类型，如果长度不同也会导致未定义的行为，如：

```
void bar(int n)
{
    int a[5];
    typedef int t[n];     // Non-compliant, a variable length array type
    t* p = &a;            // Undefined behavior if n != 5
    ....
}
```

规则 5.7.2.56：避免使用在栈上分配内存的函数

alloca、strdupa 等函数可以在栈上动态分配内存，但分配失败时的行为不受程序控制。

示例：

```
#include <alloca.h>


void fun(size_t n) {
    int* p = (int*)alloca(n * sizeof(int));    // Non-compliant
    if (!p) {
        return;    // Invalid
    }
    ....
}
```

例中 alloca 函数在失败时往往会使程序崩溃，对其返回值的检查是无效的。这种后果不可控的函数应避免使用，尤其在循环和递归调用过程中更不应使用这种函数。

规则 5.7.2.57：局部数组不应过大

局部数组在栈上分配空间，如果占用空间过大会导致栈溢出错误。

应关注具有较大数组的函数，评估其在运行时的最大资源消耗是否符合执行环境的要求。

示例：

```
void foo() {
    int arr[1024][1024][1024];      // Non-compliant, too large
    ....
}
```

在栈上分配空间难以控制失败情况，如果条件允许可改在堆上分配：

```
void foo() {
    int* arr = (int*)malloc(1024 * 1024 * 1024 * sizeof(int));      // Compliant
    if (arr) {
```

```
        ....        // Normal procedure
    } else {
        ....        // Handle allocation failures
    }
}
```

规则 5.7.2.58：避免不必要的内存分配

对单独的基本变量或只包含少量基本变量的对象不应使用动态内存分配。

示例：

```
bool* pb = new bool;     // Non-compliant
char* pc = new char;     // Non-compliant
```

内存分配的开销远大于变量的直接使用，而且还涉及到回收问题，是得不偿失的。

应改为：

```
bool b = false;     // Compliant
char c = 0;         // Compliant
```

用 new 分配数组时方括号被误写成小括号，或使用 unique_ptr 等智能指针时遗漏了数组括号也是常见笔误，如：

```
int* pi = new int(32);                  // Non-compliant
auto ui = make_unique<int>(32);     // Non-compliant
```

应改为：

```
int* pi = new int[32];                  // Compliant
auto ui = make_unique<int[]>(32);     // Compliant
```

有时可能需要区分变量是否存在，用空指针表示不存在，并通过资源分配创建变量的方式属于低效实现，不妨改用变量的特殊值表示变量的状态，在 C++ 代码中也可使用 std::optional 实现相关功能。

规则 5.7.2.59：避免动态内存分配

标准库提供的动态内存分配方法，其算法或策略不在使用者的控制之内，很多细节是标准没有规定的，而且也是内存耗尽等问题的根源，有高可靠性要求的嵌入式系统应避免动态内存分配。

在内存资源有限的环境中，由于难以控制具体的分配策略，很可能会导致已分配的空间用不上，未分配的空间不够用的情况。而在资源充足的环境中，也应尽量避免动态分配，如果能在栈上创建对象，就不应采用动态分配的方式，以提高效率并降低资源管理的复杂性。

示例：

```
void foo() {
    std::vector<int> v;     // Non-compliant
    ....
}
```

例中 vector 容器使用了动态内存分配方法，容量的增长策略可能会导致内存空间的浪费，甚至使程序难以稳定运行。

规则 5.7.2.60：判断资源分配函数的返回值是否有效

malloc 等函数在分配失败时返回空指针，如果不加判断直接使用会导致标准未定义的行为。

在有虚拟内存支持的平台中，正常的内存分配一般不会失败，但申请内存过多或有误时（如参数为负数）也会导致分配失败，而对于没有虚拟内存支持的或可用内存有限的嵌入式系统，检查分配资源是否成功是十分重要的，所以本规则应该作为代码编写的一般性要求。

库的实现更需要注意这一点，如果库由于分配失败而使程序直接崩溃，相当于干扰了主程序的决策权，很可能会造成难以排查的问题，对于有高可靠性要求的软件，在极端环境中的行为是需要明确设定的。

示例：

```
void foo(size_t n) {
    char* p = (char*)malloc(n);
    p[n - 1] = '\0';                    // Non-compliant, check ‘p’ first
    ....
```

```
}
```

示例代码未检查 p 的有效性便直接使用是不符合要求的。