C/C++ 安全规则集合 version 1.1.0

> Bjarne Stroustrup: "*C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows your whole leg off.*"

针对 C、C++ 语言，本文收录了 417 种需要重点关注的问题，可为制定编程规范提供依据，也可为代码审计以及相关培训提供指导意见，适用于桌面、服务端以及嵌入式等软件系统。

每个问题对应一条规则，每条规则可直接作为规范条款或审计检查点，本文是适用于不同应用场景的规则集合，读者可根据自身需求从中选取某个子集作为规范或审计依据，从而提高软件产品的安全性。

规则说明

规则按如下主题分为 15 个类别：

1. Security：敏感数据保护、攻击防御等问题
2. Resource：资源分配、使用与回收
3. Precompile：预处理指令、宏、注释等问题
4. Global：全局及命名空间作用域相关问题
5. Type：类型相关的设计与实现
6. Declaration：声明
7. Exception：异常
8. Function：函数实现
9. Control：流程控制
10. Expression：表达式
11. Literal：常量
12. Cast：类型转换
13. Buffer：缓冲区
14. Pointer：指针
15. Style：样式、风格等问题

每条规则包括：

- 编号：规则在本文中的章节编号，以"R"开头，称为 Section-ID
- 名称：用简练的短语描述违反规则的状况，以"ID_"开头，称为 Fault-ID
- 标题：规则的定义
- 说明：规则设立的原因、示例、违反规则的后果、改进建议、参照依据、参考资料等内容

当代码出现违反规则的情况时，可分为：

- Error：可直接导致错误的问题
- Warning：可导致错误或存在隐患的问题
- Suspicious：代码的可疑形式
- Suggestion：对提高代码质量的建议

规则的说明包含：

- 示例：规则相关的示例代码，指明符合规则（Compliant）的和违反规则（Non-compliant）的情况
- 相关：与当前规则有相关性的规则，可作为扩展阅读的线索
- 依据：规则依照的 ISO/IEC 标准，C 规则以 ISO/IEC 9899:2011 为主，C++ 规则以 ISO/IEC 14882:2011 为主
- 配置：某些规则的对象可由用户指定，审计工具可以此为参照实现定制化功能
- 参考：规则参考的其他规范条款，如 C++ Core Guidelines、MISRA、CWE、SEI CERT 等，也可作为扩展阅读的线索

规则的相关性分为：

- 特化：设规则 A 的特殊情况需要由规则 B 阐明，称规则 B 是规则 A 的特化
- 泛化：与特化相反，称规则 A 是规则 B 的泛化
- 相交：设两个规则针对不同的问题，但在内容上有一定的交集，称这两个规则相交

规则以“`标准名称:版本 章节编号(段落编号)-性质`”的格式引用标准，如“`ISO/IEC 14882:2011 5.6(4)-undefined`”，表示引用 C++11 标准的第 5 章第 6 节第 4 段说明的具有 undefined 性质的问题。

其中“性质”分为：

- undefined：一般指某种错误，使程序产生 undefined behavior
- unspecified：标准不作明确规定的情况，由编译器或环境自主定义，具有随意性
- implementation defined：由实现定义，也是由编译器或环境自主定义，与 unspecified 不同，要求有明确的文档支持
- deprecated：已过时或已废弃的用法

本文以 ISO/IEC 9899:2011、ISO/IEC 14882:2011 为主要依据，兼顾 C18、C++17 以及历史标准，没有特殊说明的规则同时适用于 C 语言和 C++ 语言，只适用于某一种语言的规则会另有说明。

规则选取

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。

如果将本文作为培训内容，为了全面理解各种场景下存在的问题，应选取全部规则。

规则列表

1. Security

R1.1 敏感数据不可写入代码

ID_plainSensitiveInfo　　security warning

代码中的敏感数据极易泄露，产品及相关运维、测试工具的代码均不可记录任何敏感数据。

示例：

```
/**
 * My name is Rabbit
 * My passphrase is Y2Fycm90         // Non-compliant
 */

#define PASSWORD "Y2Fycm90"          // Non-compliant

const char* passcode = "Y2Fycm90";   // Non-compliant
```

将密码等敏感数据写入代码是非常不安全的，即使例中 Y2Fycm90 是实际密码的某种变换，聪明的读者也会很快将其破解。

敏感数据的界定是产品设计的重要环节。对具有高可靠性要求的客户端软件，不建议保存任何敏感数据，对于必须保存敏感数据的软件系统，则需要落实安全的存储机制以及相关的评审与测试。

相关

ID_secretLeak

参考

CWE-798
CWE-259
SEI CERT MSC41-C

R1.2 敏感数据不可被系统外界感知

ID_secretLeak　　security warning

敏感数据出入软件系统时需采用有效的保护措施。

示例：

```
void foo(User* u) {
    log("Messages for %s and %s", u->name, u->password);  // Non-compliant
}
```

显然，将敏感数据直接输出到界面、日志或其他外界可感知的介质中是不安全的，需避免敏感数据的有意外传，除此之外，还需要落实具体的保护措施。

保护措施包括但不限于：

- 避免用明文或弱加密方式传输敏感数据
- 避免敏感数据从内存交换到外存
- 避免如除零、无效指针解引用等问题造成“core dump”
- 应具备反调试机制，使外界无法获得进程的内部数据
- 应具备反注入机制，使外界无法篡改程序的行为

下面以 Windows 平台为例，给出阻止敏感数据从内存交换到外存的示例：

```
class SecretBuf {
    size_t len = 0;
    unsigned char* buf = nullptr;

public:
    SecretBuf(size_t size) {
        auto* tmp = (unsigned char*)VirtualAlloc(
            0, size, MEM_RESERVE | MEM_COMMIT, PAGE_READWRITE
        );
        if (VirtualLock(tmp, size)) {  // The key point
            buf = tmp;
            len = size;
        } else {
            VirtualFree(tmp, 0, MEM_RELEASE);
        }
```

```
    }

   ~SecretBuf() {
        SecureZeroMemory(buf, len);     // Clear the secret content
        VirtualUnlock(buf, len);
        VirtualFree(buf, 0, MEM_RELEASE);
        len = 0;
        buf = nullptr;
    }

    size_t size() const { return len; }
    unsigned char* ptr() { return buf; }
    const unsigned char* ptr() const { return buf; }
};
```

例中 SecretBuf 是一个缓冲区类，其申请的内存会被锁定在物理内存中，不会与外存交换，可在一定程度上防止其他进程的恶意嗅探，保障缓冲区内数据的安全。SecretBuf 在构造函数中通过 VirtualLock 锁定物理内存，在析构函数中通过 VirtualUnlock 解除锁定，解锁之前有必要清除数据，否则解锁之后残留数据仍有可能被交换到外存，进一步可参见 ID_unsafeCleanup。

SecretBuf 的使用方法如下：

```
void foo() {
    SecretBuf buf(256);
    if (buf.ptr()) {
        // ... Do something secret using buf.ptr() ...
    } else {
        // ... Handle memory error ...
    }
}
```

对于 Linux 等系统可参见如下有相似功能的系统 API：

```
int mlock(const void* addr, size_t len);     // In <sys/mman.h>
int munlock(const void* addr, size_t len);
int mlockall(int flags);
int munlockall(void);
```

相关

ID_unsafeCleanup

参考

CWE-528
CWE-591
SEI CERT MEM06-C
SEI CERT MEM06-CPP

R1.3 敏感数据在使用后应被有效清理

ID_unsafeCleanup　　security warning

及时清理不再使用的敏感数据是重要的安全措施，且应保证清理过程不会因为编译器的优化而失效。

示例：

```
void foo() {
    char user[32] = {};
    char password[32] = {};
    if (get_input(user, password, 32)) {
        login(user, password);
    }
    memset(password, 0, sizeof(password));  // Non-compliant
}
```

本例调用 memset 覆盖敏感数据以达到清理的目的，然而敏感信息 password 为局部数组且 memset 之后没有再被引用，根据相关标准，编译器可将 memset 过程去掉，结果是没有得到有效清理。

C11 提供了 memset_s 函数以避免这种问题，某些平台和库也提供了相关支持，如 SecureZeroMemory、explicit_bzero、OPENSSL_cleanse 等不会被优化掉的函数。

对于 C++ 语言，可将敏感数据地址设为 volatile 以避免编译器的优化，再用 std::fill 等方法清理，如：

```
void foo() {
    ....
    volatile char  v_padding = 0;
    volatile char* v_address = password;
    std::fill_n(v_address, sizeof(password), v_padding);   // Compliant
}
```

也可以仿照 std::fill_n 利用 volatile 关键字的特性自行实现相关功能，可参见 ID_forbidVolatile。

关于本规则的实际应用可参见 ID_secretLeak 的相关示例。

相关

ID_secretLeak
ID_forbidVolatile

依据

ISO/IEC 9899:2011 5.1.2.3(4)
ISO/IEC 9899:2011 K.3.7.4.1

参考

CWE-14
CWE-733
SEI CERT MSC06-C

R1.4 公共成员或全局对象不应记录敏感数据

ID_sensitiveName　　security warning

公共成员、全局对象可被外部代码引用，如果存有敏感数据则可能会被误用或窃取。

示例：

```
extern string password;   // Non-compliant

struct A {
    string username;
    string password;      // Non-compliant
};
```

应至少将相关成员改为私有：

```
class A {
public:
    // ... interfaces for accessing passwords safely
private:
    string username;
    string password;      // Compliant
};
```

敏感数据最好对引用者完全隐藏，使数据与接口进一步分离，可参见“Pimpl idiom”等模式。

参考

CWE-766

R1.5 预判用户输入造成的不良后果

ID_hijack　　security warning

须对用户输入的脚本、路径、资源请求等信息进行预判，对产生不良后果的输入予以拒绝。

示例：

```
Result foo() {
    return sqlQuery(
        "select * from db where key='%s'", userInput()  // Non-compliant
    );
}
```

设 userInput 返回用户输入的字符串，sqlQuery 将用户输入替换格式化占位符后执行 SQL 语句，如果用户输入“xxx' or 'x'='x”一类的字符串则相当于执行的是“select * from db where key='xxx' or 'x'='x'”，一个恒为真的条件使 where 限制失效，造成所有数据被返回，所以在执行 SQL 语句之前应判断用户输入的安全性。

又如：

```
string bar() {
    return readFile(
        "/myhome/mydata/" + userInput()  // Non-compliant
    );
}
```

这段代码意在将用户输入的路径限制在 /myhome/mydata 目录下，然而这么做是不安全的，如果用户输入带有“../”这种相对路径，则仍可绕过限制，所以在读取文件之前应判断路径的可靠性。

注意，“用户输入”不单指人的输入，不受程序直接控制的数据，如源自外存、硬件或其他进程的输入均在此范围内。

参考

CWE-89
CWE-23
CWE-73

R1.6 避免在一个事务中通过路径多次访问同一文件

ID_TOCTOU　　security warning

攻击者可以在两次通过路径访问文件的中途对文件做手脚，从而造成不良后果。

这种问题称为“TOCTOU（Time-of-check to time-of-use）”。有时需要先检查文件的某种状态，如果状态满足条件的话，再使用该文件，如果“检查”和“使用”都是通过路径完成的，攻击者可以在中途将文件替换成不满足条件的文件，如将文件替换成指向另一个文件的链接，从而对系统造成破坏。

示例：

```
void create(const char* path) {
    FILE* fp = fopen(path, "r");
    if (fp != NULL) {              // #1, time-of-check
        fclose(fp);
        return;
    }
    fp = fopen(path, "w");         // #2, time-of-use, non-compliant
    if (fp != NULL) {
        fwrite("abc", 1, 3, fp);
        fclose(fp);
    }
}
```

示例代码先通过路径判断文件是否存在，如果存在则不作处理，如果不存在则再次通过路径创建文件并写入数据。如果攻击者把握住时机，在程序执行到 #1 和 #2 之间时按 path 创建指向其他文件的链接，那么被指向的文件会遭到破坏，尤其是当被攻击的进程权限比较高时，破坏力是难以控制的。

应只通过路径打开文件对象一次，只通过文件对象操作文件：

```
void create(const char* path) {
    FILE* fp = fopen(path, "wx");  // Compliant, since C11
    if (fp != NULL) {
        fwrite("abc", 1, 3, fp);
        fclose(fp);
    }
}
```

利用“wx”模式即可保证 fopen 在文件不存在时创建文件，文件存在时返回空。

注意，目前 C++ 的 fstream 尚无法完成与“wx”模式相同的功能，相同功能的代码要用 fopen 实现。

依据

ISO/IEC 9899:2011 7.21.5.3(3)

参考

CWE-367

R1.7 访问共享数据应遵循合理的同步机制

ID_dataRaces　　security warning

共享数据可被多个执行单位或硬件读写，需要合理控制访问的先后顺序。

示例：

```
int foo() {
    static int id = 0;
    return id++;        // Data races in multithreading
}
```

例中 foo 函数意在返回不同的整数，但如果 id 被多个线程同时读写会导致标准未定义的行为，得到错误的结果。

应改为：

```
int foo() {
    static atomic<int> id(0);
    return id.fetch_add(1);    // OK
}
```

其中 atomic 是 C++ 标准原子类，fetch_add 将对象持有的整数增 1 并返回之前的值，这个过程不会被多个线程同时执行，只能依次执行，从而保证了返回值的唯一性。

又如：

```
void bar() {
    int* p = baz();      // #0, 'p' points to shared data
    if (*p == 0) {       // #1, '*p' is unreliable
        ....
    }
    else if (*p == 1) {  // #2, '*p' is unreliable
        ....
    }
    else {               // #3
        ....
    }
}
```

如果 p 指向共享数据，那么攻击者可以通过控制共享数据实现对程序流程的劫持，比如在 #0 处 *p 的值本为 0，攻击者在 #1 之前改变 *p 的值，迫使流程向 #2 或 #3 处跳转。

如果程序的正确性依赖进线程处理数据的特定时序，一旦这种特定时序被打破，便会产生错误和漏洞，这种情况称为"竞态条件（race conditon）"，攻击者可以抢在某关键过程前后通过修改共享数据达到攻击目的，所以应合理设计数据的访问方式或使用锁、信号量等同步手段保证数据的可靠性。

依据

ISO/IEC 9899:2011 5.1.2.4(3)-undefined
ISO/IEC 9899:2011 5.1.2.4(20)-undefined
ISO/IEC 9899:2011 5.1.2.4(25)-undefined

参考

CWE-362
C++ Core Guidelines CP.2

R1.8 对文件设定合理的权限

ID_unlimitedAuthority　　security warning

文件的访问权限不可过于宽松，否则很容易遭到窃取或篡改。

示例：

```
umask(000);  // Non-compliant
FILE* fp = fopen("bar", "w");  // Old method
....
fclose(fp);
```

例中 umask 函数开放了所有用户对文件的读写权限，这是很不安全的，进程之间不应直接通过文件通信，应实现安全的接口和交互机制。

由于历史原因，C 语言的 fopen 和 C++ 语言的 fstream 都不能确保文件只能被当前用户访问，C11 提供了 fopen_s，C++17 提供了 std::filesystem::permissions 以填补这方面的需求。

fopen_s 简例：

```
FILE* fp = NULL;
errno_t e = fopen_s(&fp, "bar", "w");  // Good
....
fclose(fp);
```

与 fopen 不同，fopen_s 可以不受 umask 等函数的影响，直接将文件的权限设为当前用户私有，是一种更安全的方法。

依据

ISO/IEC 9899:2011 K.3.5.2.1(7)
ISO/IEC 14882:2017 30.10.15.26

参考

CWE-266
CWE-732
SEI CERT FIO06-C

R1.9 落实对用户的权限管理

ID_improperAuthorization　　security warning

需落实对用户的权限管理，对无权限的请求予以拒绝。

示例：

```
Result foo() {
    auto req = getRequest();
    auto res = sqlQuery(
        "select * from db where key='%s'", req["key"]
    );
    return res;
}
```

设 req 对应用户请求，sqlQuery 将请求中的 key 字段替换格式化占位符后执行查询，这个模式存在多种问题，应判断用户是否具有读取数据库相关字段的权限，而且还应判断 req["key"] 的值是否安全，详见 ID_hijack。

参考

CWE-285

R1.10 不应引用危险符号名称

ID_dangerousName　　security warning

弱加密、弱哈希、弱随机、不安全的协议等相关库、函数、类、宏、常量等名称不应出现在代码中。

这种危险符号名称主要来自：

- 低质量随机数生成算法，如 srand、rand 等
- 不再适用的哈希算法，如 MD2、MD4、MD5、MD6、RIPEMD 以及 SHA-1 等
- 非加密协议，如 HTTP、FTP 等
- 低版本的传输层安全协议，如 TLSv1.2 之前的版本
- 弱加密算法，如 DES、3DES 等

示例：

```
#include <openssl/md5.h>   // Non-compliant

const string myUrl = "http://foo/bar";  // Non-compliant, use https

void foo() {
    MD5_CTX c;       // Non-compliant
    MD5_Init(&c);    // Non-compliant
    ....
```

```
}

void bar() {
    srand(0);        // Non-compliant
    EVP_des_ecb();   // Non-compliant
    ....
}
```

审计工具不妨通过配置设定关注的危险符号名称，当代码中出现了这些名称时就给出警告。

配置示例：

```
[ID_dangerousName]
srand|random_shuffle=Weak random
EVP_des_ecb|EVP_des_cbc=Weak encryption
http|ftp=Non encrypted protocol
CURL_SSLVERSION_TLSv1|TLS1_1_VERSION=Old TLS version
```

配置项等号左侧为危险符号名称，右侧为相关说明，可以指定多个名称对应一个说明，用“|”分隔。

例中 srand 以及弱加密算法 EVP_des_ecb、EVP_des_cbc 等名称被设为危险名称，当代码中出现同名符号时就按相关说明给出警告。

配置

详见说明

参考

CWE-326
CWE-327

R1.11 避免调用具有危险性的函数

ID_dangerousFunction　　security warning

某些库函数或系统 API 本身就具有危险性，使用这种函数相当于直接引入了风险。

示例：

```
gets       // The most dangerous function
mktemp     // Every use of ‘mktemp’ is a security risk, use ‘mkstemp’ instead
getpass    // Unsafe and not portable
crypt      // Unsafe, exhaustive searches of the key space are possible
getpw      // It may overflow the provided buffer, use ‘getpwuid’ instead
cuserid    // Not portable and unreliable, use ‘getpwuid(geteuid())’ instead
chgrp      // Prone to TOCTOU race conditions, use ‘fchgrp’ instead
chown      // Prone to TOCTOU race conditions, use ‘fchown’ instead
chmod      // Prone to TOCTOU race conditions, use ‘fchmod’ instead
```

```
SuspendThread       // Forced suspension of a thread can cause many problems
TerminateThread     // Forced termination of a thread can cause many problems
GlobalMemoryStatus        // Return incorrect information, use
‘GlobalMemoryStatusEx’ instead
SetProcessWorkingSetSize  // Cause adverse effects on other processes and the
entire system
```

gets 等函数无法检查缓冲区大小，是公认的危险函数，TerminateThread 等 Windows API 会强制结束线程的执行，线程持有的资源无法正确释放会导致泄漏或死锁，这类函数应避免使用。

审计工具不妨通过配置设定关注的危险函数名称，当代码中出现了这些名称时就给出警告。

配置示例：

```
[ID_dangerousFunction]
gets|_getws=The most dangerous function
TerminateThread=Forced termination of a thread can cause many problems
```

配置项等号左侧为危险函数名称，右侧为相关说明，可以指定多个名称对应一个说明，用“|”分隔。

配置

详见说明

参考

CWE-242
CWE-474
CWE-676

R1.12 不应调用已过时的函数

ID_obsoleteFunction　　security warning

某些库函数或系统 API 存在缺陷并已宣布过时，应使用更完善的替代方法。

示例：

```
ctime           // Use ‘strftime’ instead
asctime         // Use ‘strftime’ instead
bcmp            // Use ‘memcmp’ instead
bcopy           // Use ‘memmove’ or ‘memcpy’ instead
bsd_signal      // Use ‘sigaction’ instead
gethostbyaddr   // Use ‘getnameinfo’ instead
gethostbyname   // Use ‘getaddrinfo’ instead

RegCreateKey    // Use ‘RegCreateKeyEx’ instead
RegEnumKey      // Use ‘RegEnumKeyEx’ instead
RegOpenKey      // Use ‘RegOpenKeyEx’ instead
RegQueryValue   // Use ‘RegQueryValueEx’ instead
RegSetValue     // Use ‘RegSetValueEx’ instead
```

例中 C89 声明的 ctime、asctime 等函数在 POSIX.1-2008 中已宣告过时，应改用 strftime，RegCreateKey 等 16 位 Windows API 在 32 和 64 位平台中不应再使用。

审计工具不妨通过配置设定关注的过时函数名称，当代码中出现了这些名称时就给出警告。

配置示例：

```
[ID_obsoleteFunction]
asctime|asctime_r=use 'strftime' instead
bcopy=use 'memmove' or 'memcpy' instead
```

配置项等号左侧为过时函数名称，右侧为相关说明，可以指定多个名称对应一个说明，用“|”分隔。

配置

详见说明

参考

CWE-477

R1.13 禁用不安全的字符串函数

ID_unsafeStringFunction　　⛔ security warning

由于历史原因，C 语言某些字符串函数不检查缓冲区长度，易造成运行时错误或安全漏洞。

这类函数包括：

```
gets、sprintf、scanf、sscanf、fscanf、
vfscanf、vsprintf、vscanf、vsscanf、
strcpy、strcat、wcscpy、wcscat、
StrCpy、StrCpyA、StrCpyW、StrCat、StrCatA、StrCatW、
lstrcat、lstrcatA、lstrcatW、lstrcpy、lstrcpyA、lstrcpyW
```

对于 C++ 代码，应采用 STL 标准库提供的相关功能。
对于 C 代码，应采用更安全的库函数，如用 fgets 代替 gets，snprintf 代替 sprintf。

示例：

```
char buf[100];
gets(buf);      // Non-compliant
```

例中 gets 函数无法检查缓冲区的大小，一旦输入超过了 buf 数组的边界，程序的数据或流程就会遭到破坏，这种情况也会成攻击者的常用手段，可参见 ID_bufferOverflow 的进一步说明。如果代码中存在 gets 等函数，可以直接判定程序是有漏洞的。

应改为：

```
char buf[100];
fgets(buf, sizeof(buf), stdin);  // Compliant
```

fgets 与 gets 不同，当输入超过缓冲区大小时会被截断，保证缓冲区之外的数据不会被破坏。

又如：

```
char buf[100];
scanf("%s", buf);  // Non-compliant
```

例中 scanf 函数与 gets 函数有相同的问题，可改为：

```
char buf[100];
scanf("%99s", buf);  // Let it go, but ‘fgets’ is better
```

scanf、sprintf、strcpy 等函数无视缓冲区大小，需要在外部另行实现防止缓冲区溢出的代码，完全依赖于编写者的小心谨慎。历史表明，对人的单方面依赖是不可靠的，改用更安全的方法才是明智的选择。

相关

ID_bufferOverflow

依据

ISO/IEC 9899:2011 K.3.7

参考

CWE-119
CWE-120
CWE-676
MISRA C++ 2008 18-0-5

R1.14 确保字符串以空字符结尾

ID_improperNullTermination　　security warning

语言要求字符串以空字符结尾，程序应保证有足够的内存空间安置空字符，否则会破坏程序基本的执行机制，造成严重问题。

空字符指 '\0'、L'\0'、u'\0'、U'\0'，分别对应 char*、wchar_t*、char16_t*、char32_t* 等字符串类型。

示例：

```
void foo(const char* p) {
    char a[4];
    strncpy(a, p, sizeof(a));
    printf("%s\n", strupr(a));  // To upper case and print, dangerous
}
```

例示代码将字符串复制到数组中，转为大写并打印，然而如果 p 所指字符串的长度超过 3，strncpy 不会在数组的结尾安置空字符 '\0'，会导致内存访问错误。

应改为：

```
void foo(const char* p) {
    char a[4] = "";                 // Initialize all to '\0'
    strncpy(a, p, sizeof(a));
    if (a[3] == '\0') {
        printf("%s\n", strupr(a));  // OK
    } else {
        ....                        // Handle string length exceptions
    }
}
```

将所有数组元素初始化为 '\0'，调用 strncpy 后如果数组最后一个元素是 '\0'，说明输入字符串的长度符合要求，否则可作出相应的异常处理。

相关

ID_unsafeStringFunction

依据

ISO/IEC 9899:2011 7.24.2.4

参考

CWE-170

R1.15 避免使用由实现定义的库函数

ID_implementationDefinedFunction　　security warning

由实现定义的（implementation-defined）库函数会增加移植或兼容等方面的成本。

如：

- cstdlib、stdlib.h 中的 abort、exit、getenv 或 system 等函数
- ctime、time.h 中的 clock 等函数
- csignal、signal.h 中的 signal 等函数

这些函数的行为取决于编译器、库或环境的生产厂家，同一个函数不同的厂家会有不同的实现，故称这种函数的行为是“由实现定义”的。有高可靠性要求的软件系统应避免使用这种函数，否则需明确各种实现上的具体差异，提高了移植、发布以及兼容性等多方面的成本。

示例：

```
#include <cstdlib>

void foo() {
    abort();  // Non-compliant
}
```

标准规定调用 abort 后进程应被终止，但进程打开的流是否会被关闭，创建的临时文件是否会被清理等问题没明确定义。

依据

ISO/IEC 9899:2011 7.22.4.1(2)-implementation
ISO/IEC 9899:2011 7.22.4.4(5)-implementation
ISO/IEC 9899:2011 7.22.4.6(2)-implementation
ISO/IEC 9899:2011 7.22.4.8(3)-implementation
ISO/IEC 9899:2011 7.27.2.1(3)-implementation
ISO/IEC 9899:2011 7.14.1.1(3)-implementation

参考

MISRA C 2004 20.8
MISRA C 2004 20.11
MISRA C 2004 20.12
MISRA C 2012 21.5
MISRA C 2012 21.8
MISRA C 2012 21.10
MISRA C++ 2008 18-0-3
MISRA C++ 2008 18-0-4
MISRA C++ 2008 18-7-1

R1.16 除数的值不可为 0

ID_divideByZero　　security error

除数为 0 会使程序产生标准未定义的行为。

示例：

```
int foo(int n) {
    if (n) {
        ....
    }
    return 100 / n;  // Non-compliant, must determine whether ‘n’ is 0
}
```

当除数为 0 时，对于整形数据的除法，进程往往会崩溃，对于浮点型数据的除法，一般会产生“NaN”这种无效的结果。

崩溃会给用户不好的体验，而且要注意如果崩溃可由外部输入引起，会被攻击者利用从而迫使程序无法正常工作，具有高可靠性要求的服务类程序更应该注意这一点，可参见“拒绝服务攻击”的进一步说明。对于客户端程序，也要防止攻击者对崩溃产生的“core dump”进行恶意调试，避免泄露敏感数据，总之程序的健壮性与安全性是紧密相关的。

依据

ISO/IEC 9899:1999 6.5.5(5)-undefined
ISO/IEC 9899:2011 6.5.5(5)-undefined
ISO/IEC 14882:2011 5.6(4)-undefined
ISO/IEC 14882:2017 8.6(4)-undefined

参考

CWE-369
C++ Core Guidelines ES.105

R1.17 禁用 atof、atoi、atol 以及 atoll 等函数

ID_forbidAtox　　security warning

当字符串无法被正确转为数值时，stdlib.h 或 cstdlib 中的 atof、atoi、atol 以及 atoll 等函数存在标准未定义的行为。

对于 C 语言应改用 strtof、strtol 等函数，对于 C++ 语言应改用标准流转换的方式。

示例：

```
cout << atoi("abcdefg") << '\n';  // Non-compliant
cout << atoi("100000000000") << '\n';  // Non-compliant
```

例中字符串“abcdefg”不表示数字，字符串“100000000000”超出了正常 int 型变量的范围，这些情况会导致标准未定义的问题。

更严重的问题是无法通过这种函数判断转换是否成功，这种不确定性也意味着代码在实现上存在缺陷。

C++ 标准流转换示例：

```
int foo(const char* s) {
    int v = 0;
    stringstream ss(s);
    ss >> v;
    if (ss.fail()) {  // Or use ‘!ss.eof() || ss.fail()’
        throw some_exception();
    }
    return v;
}
```

本例通过 ss.fail() 判断字符串前面的字符是否可以转为 int 型变量，也可通过 !ss.eof() || ss.fail() 判断字符串整体是否可以转为 int 型变量。

依据

ISO/IEC 9899:1999 7.20.1(1)-undefined
ISO/IEC 9899:2011 7.22.1(1)-undefined

参考

CWE-190
MISRA C 2004 20.10
MISRA C 2012 21.7
MISRA C++ 2008 18-0-2

R1.18 格式化字符串应为常量

ID_variableFormatString　　security warning

出于可读性和安全性的考量，格式化字符串最好直接写成常量字符串的形式。

本规则是 ID_hijack 的特化。

示例：

```
int a, b, c;
const char* fmt = foo();
....
printf(fmt, a, b, c);  // Non-compliant
```

例中格式化字符串 fmt 是变量，这种方式可读性较差，而且要注意如果 fmt 可受外界影响，则可能被攻击者利用造成不良后果。

应将 fmt 改为常量：

```
printf("%d %d %d", a, b, c);  // Compliant
```

相关

ID_hijack

参考

CWE-134

R1.19 与程序实现相关的信息不可被外界感知

ID_addressExposure　　security warning

函数或对象的地址、缓冲区的地址和长度等信息不可被外界感知，否则会成为攻击者的线索。

示例：

```
int foo(int* p, int n) {
    if (n >= some_value) {
        log("messages for %p and %d", p, n);  // Non-compliant
    }
}
```

这种代码多以调试为目的，应合理控制这种代码，不应将其编译到产品的正式版本中。

程序采用的协议、算法以及网络结构等信息也不应暴露，如果一定要存储非用户关注的信息，需要落实可靠的加密机制。

相关

ID_hardcodedIP

参考

CWE-200

R1.20 IP 地址不应写入代码

ID_hardcodedIP　　security warning

在代码中记录 IP 地址不利于维护和移植，也容易暴露产品的网络结构，属于安全隐患。

示例：

```
string host = "10.16.25.93";  // Non-compliant
foo("172.16.10.36:8080");     // Non-compliant
bar("https://192.168.73.90/index.html");  // Non-compliant
```

应从配置文件中获取 IP 地址，并配以加密措施：

```
MyConf cfg;
string host = cfg.host();     // Compliant
foo(cfg.port());              // Compliant
bar(cfg.url());               // Compliant
```

某些特殊的 IP 地址可以被排除：

```
0.0.0.0
255.255.255.255
127.0.0.1-127.255.255.255
```

相关

ID_addressExposure

R1.21 避免使用 errno

ID_deprecatedErrno　　security warning

正确使用 errno 需要注意很多繁琐的细节，极易误用。

对异常情况的错误处理往往会成为业务漏洞，使攻击者轻易地实现其目的，不应使用 errno 和与其相同的模式，应通过返回值或 C++ 异常机制来处理异常情况。

示例：

```
void foo() {
    if (somecall() == FAILED) {
        printf("somecall() failed\n");
        if (errno == SOME_VALUE) {  // ‘errno’ may have been changed by the
printf
            ....
        }
    }
}

void bar() {
    somecall0();
    somecall1();
    somecall2();
    if (errno) {  // ‘errno’ may come from any of the above functions
        ....
    }
}

void baz(const char* s) {
    errno = 0;
    int i = atoi(s);
    if (errno) {  // Invalid, ‘atoi’ has nothing to do with ‘errno’
        ....
    }
}
```

参考

MISRA C 2004 20.5
MISRA C++ 2008 19-3-1
C++ Core Guidelines E.28

2. Resource

R2.1 不可失去对已分配资源的控制

ID_resourceLeak :drop_of_blood: resource warning

已分配资源的指针、句柄或描述符等信息不可被遗失，否则相关资源无法被访问也无法被回收，会导致资源耗尽以及死锁等问题，使程序无法正确运行。

程序需要保证资源分配与回收之间的流程可达，且不可被异常中断，所在线程也不可在中途停止。

关于内存资源，本规则特化为 ID_memoryLeak。

示例：

```
void foo(const char* path) {
    FILE* p = fopen(path, "w");
    if (cond) {
        return;  // Non-compliant, ‘p’ is lost
    }
    ....
    fclose(p);
}
```

例中 p 指向文件对象，关闭文件对象之前在某种情况下返回，造成了文件资源的遗失。

相关

ID_memoryLeak

参考

C++ Core Guidelines P.8
C++ Core Guidelines E.13

R2.2 不可失去对已分配内存的控制

ID_memoryLeak　　:drop_of_blood: resource warning

已分配内存的地址不可被遗失，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，使程序无法正确运行。

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，所在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 size) {
    char* p = (char*)malloc(size);
    if (cond) {
        return;  // Non-compliant, ‘p’ is lost
    }
    ....
    free(p);
}
```

本例由局部变量 p 记录申请的内存空间，释放之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空间了，造成了内存空间的遗失。

相关

ID_resourceLeak
ID_ownerlessResource

参考

C++ Core Guidelines P.8
C++ Core Guidelines E.13

R2.3 不可访问未初始化或已释放的资源

ID_illAccess :drop_of_blood: resource error

访问未初始化或已释放的资源属于逻辑错误，也会导致标准未定义的行为。

示例：

```
void foo(const char* path) {
    FILE* p = NULL;
    char buf[100] = {};
    if (path != NULL) {
        p = fopen(path, "rb");
    }
    fread(buf, 1, 100, p);       // Non-compliant, ‘p’ may be invalid
    ....
    fclose(p);
    ....
    fread(buf, 1, 100, p);       // Non-compliant, ‘p’ is closed
    ....
}
```

关于解引用已经释放的指针，特化为 ID_danglingDeref。
关于访问未初始化的局部对象，特化为 ID_localInitialization。

相关

ID_danglingDeref
ID_localInitialization

依据

ISO/IEC 9899:2011 7.22.3.3(2)-undefined
ISO/IEC 14882:2011 3.7.4.2(4)-undefined

参考

SEI CERT FIO46-C

R2.4 资源应接受对象化管理

ID_ownerlessResource :drop_of_blood: resource warning

使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。本文示例中的 new 和 delete 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。

参考

C++ Core Guidelines R.11
C++ Core Guidelines R.12

R2.5 资源的分配与回收方法应成对提供

ID_incompleteNewDeletePair :drop_of_blood: resource suggestion

---

资源的分配方法和相应的回收方法应在同一模块中提供。

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，使维护成本显著增加，而且 so、dll、exe 等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) {
    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。

对类等逻辑模块也有相同要求，在构造函数中分配了资源，应提供相应的析构函数，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
};
```

相关

ID_memberDeallocation
ID_crossModuleTransfer
ID_incompatibleDealloc

参考

C++ Core Guidelines R.15

R2.6 资源的分配与回收方法应配套使用

ID_incompatibleDealloc　　:drop_of_blood: resource error

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。

相关

ID_incompleteNewDeletePair

依据

ISO/IEC 9899:2011 7.22.3.3(2)-undefined
ISO/IEC 9899:2011 7.22.3.4(3)-undefined

参考

SEI CERT MEM51-CPP

R2.7 模块之间不应传递容器等对象

ID_crossModuleTransfer　　:drop_of_blood: resource warning

容器的容量可以动态变化，在模块间传递这种对象会造成分配回收方面的冲突。

具有同等功能的对象，如流、字符串、智能指针等均不应在模块间传递。

在模块间传递复杂的对象也意味着模块耦合过于紧密，不是良好的设计。如果必须在模块间传递对象，需将所有相关资源的分配与回收过程限定在同一模块内，是繁琐且不利于维护的。

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 有各自独立的堆栈，由于模版库的内联实现，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，造成严重冲突。

另外，不同的模块可能由不同的编译器生成，声明与实现的差异也会导致冲突，参见“Dll hell”。

相关

ID_incompleteNewDeletePair

R2.8 对象申请的资源应在析构函数中释放

ID_memberDeallocation　　:drop_of_blood: resource warning

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。

相关

ID_memoryLeak
ID_resourceLeak

参考

C++ Core Guidelines C.31
C++ Core Guidelines E.6

R2.9 对象被移动后不应再被使用

ID_useAfterMove :drop_of_blood: resource warning

---

std::move 宣告对象的数据即将被转移到其他对象，转移之后对象在逻辑上不再有效，不应再被使用。

示例：

```
string foo(string a) {
    string b = std::move(a);
    return a + b;  // Non-compliant
}
```

例中 a 对象的数据被转移到 b 对象，之后 a 对象不再有效，对 a 重新赋值之前访问 a 属于逻辑错误。

相关

ID_unsuitableMove

参考

SEI CERT EXP63-CPP
C++ Core Guidelines ES.56

R2.10 构造函数抛出异常需避免相关资源泄漏

ID_throwInConstructor :drop_of_blood: resource warning

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 A 的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();
    }

    // Transfer ownership, make sure no exception is thrown
    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 ‘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 块中的 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。

相关

ID_ownerlessResource
ID_multiAllocation
ID_memoryLeak

R2.11 资源不可被重复释放

ID_doubleFree :drop_of_blood: resource error

重复释放资源属于逻辑错误，也会导致标准未定义的问题。

示例：

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

依据

ISO/IEC 9899:2011 7.22.3.3(2)-undefined
ISO/IEC 14882:2011 3.7.4.2(4)-undefined

参考

CWE-415

R2.12 用 delete 释放对象不可多写中括号

ID_excessiveDelete　　:drop_of_blood: resource error

用 new 分配的对象应该用 delete 释放，不可用 delete[] 释放，否则引发标准未定义的错误。

示例：

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

相关

ID_insufficientDelete

依据

ISO/IEC 14882:2003 5.3.5(2)-undefined
ISO/IEC 14882:2011 5.3.5(2)-undefined
ISO/IEC 14882:2017 8.3.5(2)-undefined

参考

C++ Core Guidelines ES.61

R2.13 用 delete 释放数组不可漏写中括号

ID_insufficientDelete　　:drop_of_blood: resource error

用 new 分配的数组应该用 delete[] 释放，不可漏写中括号，否则引发标准未定义的错误。

示例：

```
void foo(int n) {
    auto* p = new X[n];  // n default constructed Xs
    ....
    delete p;            // Non-compliant, use ‘delete[] p;’ instead
}
```

在某些环境中，可能只有数组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，则会导致资源泄漏。

相关

ID_excessiveDelete

依据

ISO/IEC 14882:2003 5.3.5(2)-undefined
ISO/IEC 14882:2011 5.3.5(2)-undefined
ISO/IEC 14882:2017 8.3.5(2)-undefined

参考

C++ Core Guidelines ES.61

R2.14 在栈上分配的空间以及非动态申请的资源不可被释放

ID_illDealloc :drop_of_blood: resource error

释放在栈上分配的空间以及非动态申请的资源会导致标准未定义的错误。

示例：

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

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

依据

ISO/IEC 9899:2011 7.22.3.3(2)-undefined
ISO/IEC 9899:2011 7.22.3.4(3)-undefined
ISO/IEC 14882:2011 3.7.4.2(4)-undefined

参考

MISRA C 2012 22.2

R2.15 在一个表达式语句中最多使用一次 new

ID_multiAllocation :drop_of_blood: resource warning

如果表达式语句多次使用 new，一旦某个构造函数抛出异常，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。

参考

C++ Core Guidelines R.13

R2.16 流式资源对象不应被复制

ID_copiedStream :drop_of_blood: resource warning

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
```

依据

ISO/IEC 9899:1999 7.19.3(6)
ISO/IEC 9899:2011 7.21.3(6)

参考

MISRA C 2012 22.5

R2.17 避免使用在栈上分配内存的函数

ID_stackAllocation :drop_of_blood: resource warning

---

alloca、__builtin_alloca 等在栈上分配内存的函数难以控制失败时的情况，尤其在循环中更不应使用这种函数。

示例：

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

例中 alloca 函数在失败时会直接崩溃，不会返回空指针，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。

相关

ID_invalidNullCheck

参考

CWE-770
SEI CERT MEM05-C

R2.18 避免不必要的内存分配

ID_unnecessaryAllocation　　:drop_of_blood: resource warning

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
```

有时可能需要用指针指向一个变量，而指针为空时表示这个变量“不存在”，对于这种情况不妨用变量的特殊值表示变量的状态。

相关

ID_dynamicAllocation

R2.19 避免动态内存分配

ID_dynamicAllocation　　:drop_of_blood: resource warning

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。

依据

ISO/IEC 9899:1999 7.20.3
ISO/IEC 9899:2011 7.22.3

参考

MISRA C 2004 20.4
MISRA C 2012 21.3
MISRA C++ 2008 18-4-1
C++ Core Guidelines R.5

R2.20 判断资源分配函数的返回值是否有效

ID_nullDerefAllocRet　　:drop_of_blood: resource warning

---

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造成标准未定义的错误。

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，如果库在分配失败时直接崩溃或不加说明地结束进程，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：

```
char* foo(size_t n) {
    char* p = (char*)malloc(n);
    for (size_t i = 0; i < n; i++) {
        p[i] = '\0';  // Non-compliant, check ‘p’ first
    }
    return p;
}
```

示例代码未检查 p 的有效性便直接使用是不符合要求的，一旦内存分配失败就会崩溃。

依据

ISO/IEC 9899:1999 7.20.3(1)
ISO/IEC 9899:2011 7.22.3(1)

参考

CWE-476
CWE-252

R2.21 C++ 代码中禁用 C 内存管理函数

ID_forbidMallocAndFree　　⛔ resource warning

在 C++ 代码中不应使用 malloc、free 等 C 内存管理函数，应使用对象化管理方法。

示例：

```
void foo(size_t n) {
    int* p = (int*)malloc(n * sizeof(int));  // Unsafe and verbose
    ....
    free(p);
}
```

应改为：

```
void foo(size_t n) {
    auto p = make_unique<int[]>(n);  // Safe and brief
    ....
}
```

相关

ID_ownerlessResource

参考

C++ Core Guidelines R.10

3. Precompile

3.1 Include

R3.1.1 include 指令应符合标准格式

ID_illFormedInclude　　precompile error

#include 后只应为 < 头文件路径 > 或 " 头文件路径 "，否则会导致标准未定义的行为。

示例：

```
#include <string.h>         // Compliant
#include "string.h"         // Compliant

#define HEADER "string.h"
#include HEADER             // Compliant

#include stdlib.h           // Non-compliant, undefined behavior
```

例中对 string.h 的引用符合标准，而对 stdlib.h 的引用会导致标准未定义的行为。

注意，由引号标识的头文件路径并非字符串常量，不应对其使用字符串常量的特性，如：

```
#include "stdlib" ".h"      // Non-compliant, implementation defined
```

是否会将引号中的内容连接成一个路径是由实现定义的，这种代码是不可移植的。

另外，如下形式的代码也是不符合标准的：

```
#include L"foo"             // Non-compliant
#include u"bar"             // Non-compliant
#include U"baz"             // Non-compliant
#include R"(..\foo\bar)"    // Non-compliant
```

相关

ID_nonStandardCharInHeaderName

依据

ISO/IEC 14882:2011 2.9
ISO/IEC 14882:2011 16.2(4)-undefined
ISO/IEC 14882:2011 16.2(4)-implementation

参考

MISRA C 2004 19.3
MISRA C 2012 20.3
MISRA C++ 2008 16-2-6

R3.1.2 include 指令中禁用不合规的字符

ID_nonStandardCharInHeaderName　　⊖ precompile warning

字母、数字、下划线、点号之外的字符可能与文件系统存在冲突，也可能导致标准未定义的问题，不应出现在头文件和相关目录名称中。

示例：

```
#include < >           // Non-compliant
#include <"foo">       // Non-compliant
#include <foo*>        // Non-compliant
#include <?bar?>       // Non-compliant

#include <foo>         // Compliant
#include <foo.h>       // Compliant
#include <foo_bar.h>   // Compliant
```

可以用 / 作为路径分隔符，但不应出现 // 或 /*， 如：

```
#include <foo//bar.h>   // Non-Compliant, undefined behavior
#include <foo/*bar.h>   // Non-Compliant, undefined behavior
```

名称中的单引号、反斜杠在 C 及 C++03 标准中是未定义的，在 C++11 标准中是由实现定义的。

```
#include <foo'bar>     // Non-compliant
#include <foo\bar>     // Non-compliant
```

另外，由于某些平台的文件系统不区分路径大小写，建议头文件名称只使用小写字母以减少移植类问题。

依据

ISO/IEC 9899:1999 6.4.7(3)-undefined
ISO/IEC 9899:2011 6.4.7(3)-undefined
ISO/IEC 14882:2003 2.8(2)-undefined
ISO/IEC 14882:2011 2.9(2)-implementation

参考

MISRA C 2004 19.2
MISRA C 2012 20.2
MISRA C++ 2008 16-2-4

R3.1.3 include 指令中不应使用反斜杠

ID_forbidBackslashInHeaderName　　⊖ precompile warning

在 include 指令中使用反斜杠不利于代码移植，而且可能会导致标准未定义的问题。

示例：

```
#include <\r\n.h>        // Non-compliant
#include <foo\\bar.h>    // Non-compliant
#include "foo\\bar.h"    // Non-compliant

#include <foo/bar.h>     // Compliant
```

C++11 之前的标准指明反斜杠出现在尖括号或引号之间的行为是未定义的，C++11 之后则由实现定义，所以对有高可移植性要求的代码应避免使用反斜杠。

依据

ISO/IEC 9899:1999 6.4.7(3)-undefined
ISO/IEC 9899:2011 6.4.7(3)-undefined
ISO/IEC 14882:2003 2.8(2)-undefined
ISO/IEC 14882:2011 2.9(2)-implementation

参考

MISRA C++ 2008 16-2-5

R3.1.4 include 指令中不应使用绝对路径

ID_forbidAbsPathInHeaderName　　⛔ precompile warning

绝对路径使代码过分依赖编译环境，意味着项目的编译设置不完善，应使用相对路径。

示例：

```
#include "C:\\foo\\bar.h"  // Non-compliant
#include "/foo/bar.h"      // Non-compliant
```

R3.1.5 禁用不合规的头文件

ID_forbiddenHeader　　⛔ precompile warning

无意义的，行为不明确的或有不良副作用的头文件应禁用。

示例：

```
#include <tgmath.h>   // Non-compliant
#include <setjmp.h>   // Non-compliant
#include <stdbool.h>  // Non-compliant in C++
```

tgmath.h 和 ctgmath 会使用语言标准之外的技术实现某种重载效果，而且其中的部分函数名称会干扰其他标准库中的名称，setjmp.h 和 csetjmp 则包含危险的过程间跳转函数。

iso646.h、stdalign.h 以及 stdbool.h 对于 C++ 语言来说没有意义，在 C++ 代码中不应使用。

stdio.h、signal.h、time.h、fenv.h 等头文件对于有高可靠性要求的软件系统也不建议使用，这些头文件含有较多标准未声明、未定义或由实现定义的内容。

审计工具不妨通过配置设定不合规头文件的名称：

```
[ID_forbiddenHeader]
tgmath.h|ctgmath=May result in undefined behaviour
setjmp.h|csetjmp=Forbidden header
```

表示将 tgmath.h、ctgmath、setjmp.h、csetjmp 设为不合规头文件，如发现代码中有 tgmath.h，则报告“May result in undefined behaviour”，如发现代码中有 setjmp.h 或 csetjmp ，则报告“Forbidden header”。

配置

详见说明

依据

ISO/IEC 14882:2017 C.5.1(4)

参考

MISRA C 2012 21.5
MISRA C 2012 21.10
MISRA C 2012 21.11
MISRA C 2012 21.12
MISRA C++ 2008 18-7-1
MISRA C++ 2008 18-0-4
MISRA C++ 2008 27-0-1

R3.1.6 C++ 代码不应引用 C 头文件

ID_forbidCHeaderInCpp　　⊖ precompile warning

在 C++ 代码中应使用 C++ 标准头文件，stdio.h、stdlib.h 等 C 语言头文件不在 C++ 标准之内，应改用 cstdio、cstdlib 等 C++ 标准头文件。

C 标准头文件均有对应的 C++ 版本，C++ 版本提供了更适合 C++ 语言的命名空间、模板以及函数重载等功能。另外，按 C++ 惯例，语言相关的标准头文件无扩展名，自定义及平台相关的以 .h 为扩展名，遵循统一的命名规范也有必要的。

示例：

```
#include <assert.h>    // Non-compliant, use <cassert>
#include <ctype.h>     // Non-compliant, use <cctype>
#include <errno.h>     // Non-compliant, use <cerrno>
#include <float.h>     // Non-compliant, use <cfloat>
#include <limits.h>    // Non-compliant, use <climits>
#include <locale.h>    // Non-compliant, use <clocale>
#include <math.h>      // Non-compliant, use <cmath>
#include <setjmp.h>    // Non-compliant, use <csetjmp>
#include <signal.h>    // Non-compliant, use <csignal>
#include <stdarg.h>    // Non-compliant, use <cstdarg>
#include <stddef.h>    // Non-compliant, use <cstddef>
#include <stdio.h>     // Non-compliant, use <cstdio>
#include <stdlib.h>    // Non-compliant, use <cstdlib>
#include <string.h>    // Non-compliant, use <cstring>
#include <time.h>      // Non-compliant, use <ctime>
#include <wchar.h>     // Non-compliant, use <cwchar>
#include <wctype.h>    // Non-compliant, use <cwctype>
```

依据

ISO/IEC 14882:2003 D.5
ISO/IEC 14882:2011 D.5
ISO/IEC 14882:2017 D.5

参考

MISRA C++ 2008 18-0-1

3.2 Macro

R3.2.1 宏应遵循合理的命名方式

ID_macro_badName　　precompile suggestion

宏的名称应采用全大写字母的形式，非宏名称则应包含小写字母。

宏用文本处理，不受语言规则限制，易被误用，在命名方式上将其与普通代码分开可引起使用者或维护者的注意，从而有助于规避错误。

本规则是 ID_badName 的特化。

示例：

```
#define word_size 8   // Non-compliant, like a normal variable
#define WORD_SIZE 8   // Compliant
```

配置

maxWordLength：连续无大小写变化的字符个数上限，超过则报出

相关

ID_badName

参考

C++ Core Guidelines ES.32
C++ Core Guidelines ES.9

R3.2.2 不可定义具有保留意义的宏名称

ID_macro_defineReserved precompile warning

重新定义已有特殊用途的名称，会使代码陷入难以维护的境地，也会导致标准未定义的问题。

C++ 标准指明不可重新定义的宏有：

```
__cplusplus、__TIME__、__DATE__、__FILE__、__ LINE__、
__STDC__、__STDC_HOSTED__、__STDCPP_THREADS__、
__STDC_MB_MIGHT_NEQ_WC__、__STDC_VERSION__、
__STDC_ISO_10646__、__STDCPP_STRICT_POINTER_SAFETY__
```

除此之外，平台、环境、框架相关的宏也不应在代码中重新定义。

以下划线开头的名称用于表示标准库或系统的内部名称，自定义名称不应以下划线开头。

示例：

```
#define defined  // Non-compliant
```

不可重定义关键字

```
#define __GNUC__ 1  // Non-compliant
```

标识编译器或平台的宏不可在代码中写死

```
#define NDEBUG 0  // Non-compliant
```

编译优化相关的宏不可在代码中写死

```
#define assert(x) ((void)x)  // Non-compliant
```

标准库中的宏不应重新实现

```
#define new new(std::nothrow)  // Non-compliant
```

应实现为调用 new(std::nothrow) 的函数

审计工具不妨通过配置设定保留名称：

```
[ID_macro_defineReserved]
keywordAsReserved=true
NULL|NDEBUG|EOF=Reserved name should not be defined or undefined
```

表示将 NULL、NDEBUG、EOF 为设为保留名称，当在代码中发现定义了相同名称的宏时则提示“Reserved name should not be redefined or undefined”。
配置项 keywordAsReserved 为 true 表示关键字也作为保留名称，否则只认为用户设置的名称为保留名称。

配置

详见说明

相关

ID_macro_undefReserved
ID_reservedName

依据

ISO/IEC 9899:2011 7.1.3(2)-undefined
ISO/IEC 14882:2011 16.8(4)-undefined

参考

MISRA C 2012 21.1
MISRA C 2012 20.4
MISRA C++ 2008 17-0-1

R3.2.3 不可取消定义具有保留意义的宏名称

ID_macro_undefReserved　　precompile warning

取消定义已有特殊用途的宏名称，会使代码陷入难以维护的境地，也会导致标准未定义的问题。

C++ 标准指明不可取消定义的宏有：

```
__cplusplus、__TIME__、__DATE__、__FILE__、__ LINE__、
__STDC__、__STDC_HOSTED__、__STDCPP_THREADS__、
__STDC_MB_MIGHT_NEQ_WC__、__STDC_VERSION__、
__STDC_ISO_10646__、__STDCPP_STRICT_POINTER_SAFETY__
```

除此之外，平台、环境、框架相关的宏也不可被取消定义。

示例：

```
#undef __LINE__     // Non-compliant
#undef __cplusplus  // Non-compliant
#undef NDEBUG       // Non-compliant
#undef _WIN64       // Non-compliant
#undef __unix__     // Non-compliant
```

审计工具不妨通过配置设定保留名称：

```
[ID_macro_undefReserved]
keywordAsReserved=true
NULL|NDEBUG|EOF=Reserved name should not be defined or undefined
```

表示将 NULL、NDEBUG、EOF 为设为保留名称，当在代码中发现 undef 相同名称的宏时则提示“Reserved name should not be redefined or undefined”。
配置项 keywordAsReserved 为 true 表示关键字也作为保留名称，否则只认为用户设置的名称为保留名称。

相关

ID_macro_defineReserved
ID_reservedName

依据

ISO/IEC 9899:2011 7.1.3(3)-undefined
ISO/IEC 14882:2011 16.8(4)-undefined

参考

MISRA C 2012 21.1
MISRA C 2012 20.5
MISRA C++ 2008 17-0-1
MISRA C++ 2008 16-0-3

R3.2.4 可作为子表达式的宏定义应该用括号括起来

ID_macro_expNotEnclosed　　precompile warning

由于宏只做文本处理，不考虑运算符优先级等问题，可作为子表达式的宏定义应该用括号括起来，否则很可容易产生意料之外的错误。

示例：

```
#define ABS(x) (x) < 0? -(x): (x)  // Non-compliant
```

设 a 为变量，如果按如下使用方式：

```
a = ABS(a) + 1;
```

则相当于：

```
a = (a) < 0? -(a): (a) + 1;
```

这显然会造成意料之外的结果，所以 ABS 的定义应改为：

```
#define ABS(x) ((x) < 0? -(x): (x))  // Compliant
```

参考

CWE-783
MISRA C 2004 19.10
MISRA C 2012 20.7

R3.2.5 与运算符相关的宏参数应该用括号括起来

ID_macro_paramNotEnclosed　　precompile warning

由于宏只做文本处理，不考虑运算符优先级等问题，故应将宏参数用括号括起来，否则很可容易产生意料之外的错误。

示例：

```
#define SUM(a, b) (a + b)  // Non-compliant
```

应改为：

```
#define SUM(a, b) ((a) + (b))  // Compliant
```

参考

CWE-783
MISRA C++ 2008 16-0-6

R3.2.6 由多个语句组成的宏定义应该用 do-while(0) 括起来

ID_macro_stmtNotEnclosed　　precompile warning

可以作为一条语句使用的宏，且宏包含多个并列子句时，应该用“do {”和 “} while(0)”括起来，否则易造成作用域的混乱。

示例：

```
#define SWAP(a, b)\
    a ^= b; b ^= a; a ^= b    // Non-compliant
```

如果按如下使用方式：

```
if (x > y)
    SWAP(x, y);
```

展开后 b ^= a; a ^= b; 不在 if 语句的范围内，应改为：

```
#define SWAP(a, b) {\       // Compliant
    a ^= b; b ^= a; a ^= b;\
}
```

更进一步地，建议使用 do-while(0) 结构：

```
#define SWAP(a, b) do {\    // Good
    a ^= b; b ^= a; a ^= b;\
} while(0)
```

这样在使用宏时必须以分号结尾，否则无法通过编译，使宏在使用风格上与函数相同，易于阅读。

相关

ID_if_scope
ID_while_scope
ID_for_scope

参考

CWE-483

R3.2.7 宏的实参个数不可小于形参个数

ID_macro_insufficientArgs　　precompile warning

宏的实参个数小于形参个数是不符合 C/C++ 标准的，参数个数不一致必然意味着某种错误，然而在某些编译环境下却可以通过编译。

示例：

```
#define M(a, b, c)  a ## b ## c

const char* foo() {
    return M("x", "y");  // Non-compliant
}
```

在早期标准中（如 ISO 9899:1990）这种情况是未定义的，而后续标准对其进行了约束，但 MSVC 2017 等编译器不把这种问题视作编译错误，需要特别注意。

相关

ID_macro_redundantArgs

参考

CWE-628
MISRA C 2004 19.8

R3.2.8 宏的实参个数不可大于形参个数

ID_macro_redundantArgs precompile warning

宏的实参个数大于形参个数是不符合 C/C++ 标准的，参数个数不一致必然意味着某种错误，然而在某些编译环境下却可以通过编译。

示例：

```
#define M(a, b, c)  a ## b ## c

const char* foo() {
    return M("a", "b", "c", "d");  // Non-compliant
}
```

相关

ID_macro_insufficientArgs

参考

CWE-628

R3.2.9 宏参数不应有副作用

ID_macro_sideEffectArgs precompile warning

当宏参数有“副作用（side effect）”时，如果宏定义中没有或多次引用到该参数，会导致意料之外的错误。

示例：

```
#define I(a)
#define M(a) ((a) + (a))

int foo(int& a) {
    return M(++a);  // Non-compliant, returns '((++a) + (++a))'
}

void bar(int& a) {
    I(a--);         // Non-compliant, does nothing
}
```

例中 M 和 I 看起来像是函数调用，而展开后的结果却在意料之外。

相关

ID_sideEffectAssertion
ID_macro_function

参考

SEI CERT PRE31-C

R3.2.10 宏参数数量应在规定范围之内

ID_macro_tooManyParams　　precompile warning

宏参数数量过多意味着宏功能过于复杂，不利于调试，应改为函数。

配置

maxParamCount：参数个数上限，超过则报出

相关

ID_tooManyParams

R3.2.11 宏名称中不应存在拼写错误

ID_macro_misspelling precompile suggestion

宏的名称不应存在拼写错误，尤其是供他人调用的宏，错误拼写会使代码的使用者对代码的质量产生疑虑，应认真对待。

示例：

```
#define FRIST(p) p->first()  // Non-compliant, should be FIRST
```

相关

ID_misspelling
ID_literal_misspelling

R3.2.12 不应使用宏定义常量

ID_macro_const precompile suggestion

宏用于文本处理，不受作用域等语言规则限制，不应使用宏实现常量等语言层面的概念。

示例：

```
namespace U {
    #define PI 3.14F  // Non-compliant
}

namespace V {
    #define PI 3.14159L  // Non-compliant
}

namespace W {
    void fun(double PI);  // Disturbed
}
```

例中宏 PI 不受命名空间的限制，第二个宏定义会覆盖第一个宏定义，而且会干扰其他作用域中相同的名称。

应改为：

```
namespace U {
    const float PI = 3.14F;  // Compliant
}

namespace V {
    const long double PI = 3.14159L;  // Compliant
}
```

为了避免混乱，建议宏只作为 #if、#elif 等指令的条件，不参与具体的功能实现。

相关

ID_macro_typeid
ID_macro_function

参考

C++ Core Guidelines ES.31
C++ Core Guidelines Enum.1

R3.2.13 不应使用宏定义类型

ID_macro_typeid　　precompile suggestion

宏用于文本处理，不受作用域等语言规则限制，不应使用宏实现类型等语言层面的概念。

示例：

```
namespace U {
    #define MyType int  // Non-compliant
}

namespace V {
    #define MyType long  // Non-compliant
}

void foo(MyType);  // Unreliable
```

例中 MyType 的最终定义是 long，第二个宏定义会覆盖第一个宏定义，这显然是不可靠的。

相关

ID_macro_sideEffectArgs
ID_macro_const
ID_macro_function

参考

C++ Core Guidelines ES.30

R3.2.14 可由函数实现的功能不应使用宏实现

ID_macro_function precompile suggestion

宏用于文本处理，不受作用域、参数传递、重载等语言规则限制，可由函数实现的功能不应使用宏实现。

示例：

```
#define SUM(a, b) ((a) + (b))  // Non-compliant
#define SUM(a, b, c) ((a) + (b) + (c))  // Non-compliant

int foo(int a, int b) {
    return SUM(a, b);  // Error
}
```

例中宏 SUM 意在获取参数的和，但宏无法被重载，最终只有一个宏被定义， foo 函数中的宏展开会造成错误。

相关

ID_macro_sideEffectArgs
ID_macro_const
ID_macro_typeid

参考

C++ Core Guidelines ES.31
MISRA C 2004 19.7
MISRA C 2012 Dir 4.9
MISRA C++ 2008 16-0-4

R3.2.15 在 C++ 代码中不应使用宏 offsetof

ID_deprecatedOffsetof precompile suggestion

宏 offsetof 很难适用于具有 C++ 特性的类，易引发未定义的错误。

示例：

```
#include <cstddef>

struct A {
    A();
    virtual ~A();

    int i, j;
```

```
};

int foo() {
    return offsetof(A, i);  // Non-compliant, undefined behavior
}

struct B {
    static int i;
    int j;
    int fun();
};

int bar() {
    return offsetof(B, i);  // Non-compliant, undefined behavior
}

int baz() {
    return offsetof(B, fun);  // Non-compliant, undefined behavior
}
```

依据

ISO/IEC 14882:2003 18.1(5)
ISO/IEC 14882:2011 18.2(4)
ISO/IEC 14882:2017 21.2.4(1)

R3.2.16 在宏定义中由 # 修饰的参数后不应出现

ID_macro_complexConcat　　precompile warning

不同编译器对 # 和 ## 的优先级有不同的实现，在有可移植性要求的代码中不应嵌套使用，而且 ## 连接的单词数量不应超过两个。

示例：

```
#define M0(a, b) # a ## b         // Non-compliant
#define M1(a, b, c) a ## #b ## c  // Non-compliant

#define M2(a) #a              // Compliant
#define M3(a, b) M1(a ## b)  // Compliant
```

依据

ISO/IEC 14882:2003 16.3.2(2)-unspecified
ISO/IEC 14882:2003 16.3.3(3)-unspecified
ISO/IEC 14882:2011 16.3.2(2)-unspecified
ISO/IEC 14882:2011 16.3.3(3)-unspecified
ISO/IEC 14882:2017 19.3.2(2)-unspecified
ISO/IEC 14882:2017 19.3.3(3)-unspecified

参考

MISRA C 2004 19.12
MISRA C 2012 20.11
MISRA C++ 2008 16-3-1

3.3 Directive

R3.3.1 头文件不应缺少守卫

ID_missingHeaderGuard precompile warning

以 .h 或 .hpp 为扩展名的头文件应包含头文件守卫。

示例：

```
// Header file foo.h
#ifndef LIBRARY_FOO_H
#define LIBRARY_FOO_H
....
#endif
```

例中 foo.h 是“Library”模块中的头文件，宏 LIBRARY_FOO_H 即可作为它的守卫，保证头文件被重复引入也不会出现问题，守卫名称不可有重复，建议守卫名称遵循“模块名_文件名”的形式。

#pragma once 指令也可作为头文件守卫，但并不是 C/C++ 的标准方式，只是多数编译器均有支持。这种方式由编译器维护一个列表，引入头文件时，如果发现文件中有 #pragma once 指令就将文件路径加入列表，当这个文件再次被 include 时便不会加载，而宏守卫的方式仍然要对文件进行预编译，所以 #pragma once 方式在编译效率上会更高一些。

宏守卫用宏名区分头文件，所以不能有重复。宏的引入可以使相关设定更灵活，比如声明头文件之间的依赖或排斥关系，如果 bar.h 依赖 foo.h，在 #include "bar.h" 之前必须 #include "foo.h"，可在 bar.h 中设置：

```
// Header file bar.h
#ifndef LIBRARY_FOO_H
#error foo.h should be included first
#endif
```

这样如果不满足条件无法通过编译。

本规则建议使用宏守卫的方式，但 #pragma once 方法也是惯用写法，不妨通过配置项决定其是否合规。

配置

allowPragmaOnce：为 true 时 #pragma once 也可作为符合要求的头文件守卫

参考

C++ Core Guidelines SF.8
MISRA C 2004 19.15
MISRA C++ 2008 16-2-3

R3.3.2 不应出现非标准格式的预编译指令

ID_illFormedDirective　　precompile warning

非标准格式的预编译指令往往意味着错误，也会导致标准未定义的问题。

需注意：

- defined 只应作用于宏名称或括号括起来的宏名称
- defined 不应出现在宏定义中
- #if、#elif 之后应为正确的常量表达式
- #ifdef、#ifndef 之后只应为宏名称
- #else、#endif 之后应直接换行

示例：

```
#if defined M            // Compliant
#if defined(M)           // Compliant
#if defined(M == 0)      // Non-compliant, undefined behaviour

#define DEFINED defined  // Non-compliant
#if DEFINED M            // Undefined behaviour
```

例中作用于比较表达式的 defined 会导致标准未定义的行为，在 #if 条件表达式中由宏展开产生的 defined 也会导致标准未定义的行为。

又如：

```
#define M 2

int foo() {
    int x = 0;
#ifdef M
        x = M;
#elif            // Non-compliant, use ‘#else’ instead
        x = 1;
#endif M         // Non-compliant, remove ‘M’
    return x;
}
```

这种代码是不符合标准的，但可被某些编译器接受，应避免。

依据

ISO/IEC 9899:2011 6.10(1)
ISO/IEC 9899:2011 6.10.1(4)-undefined
ISO/IEC 14882:2011 16.1(4)-undefined

参考

MISRA C++ 2008 16-0-7
MISRA C++ 2008 16-0-8
MISRA C++ 2008 16-1-1

R3.3.3 不应使用非标准预编译指令

ID_nonStdDirective　　precompile warning

使用非标准预编译指令并非是问题的正规解决方法，且易造成代码移植方面的隐患。

示例：

```
#import "foo"          // Non-compliant
#include_next "bar.h"  // Non-compliant
#unknwon               // Non-compliant
```

依据

ISO/IEC 9899:1999 6.10(1)
ISO/IEC 9899:2011 6.10(1)

参考

MISRA C 2004 19.16
MISRA C 2012 20.13

R3.3.4 对编译警告的屏蔽应慎重

ID_warningDisabled　　precompile suggestion

编译器一般允许使用预编译指令屏蔽某些编译警告，但对于反映风险或安全问题的警告不应屏蔽。

示例：

```
#ifdef _MSC_VER
#pragma warning(disable: 4172)  // Non-compliant
#elif defined __GNUC__
#pragma GCC diagnostic ignored "-Wreturn-local-addr"  // Non-compliant
#endif
```

示例代码屏蔽了 Visual Studio C4172 和 GCC -Wreturn-local-addr 对应的警告，当局部变量的地址被返回时编译器不会给出警告，但这种警告是不应该被屏蔽的，详见 ID_localAddressFlowOut。

本规则集合提到的部分问题编译器也可以给出警告，这种警告均不应被屏蔽。

相关

ID_warningDefault

参考

SEI CERT MSC00-C

R3.3.5 在高级别的警告设置下编译

ID_warningDefault　　precompile suggestion

编译器一般允许设定编译警告的级别，级别越高关注的问题就越多，也可以将警告设为错误，当有警告产生时停止编译，建议代码在高级别的警告设置下编译。

应避免代码中出现 #pragma warning(default:...) 等指令，这种指令将警告级别设为默认，可能与整个项目的设置不一致，如果一定要使用，应改用 #pragma warning(pop) 方式。

示例：

```
#pragma warning(disable:4706)
#include "somecode"
#pragma warning(default:4706)  // Non-compliant
```

示例代码在导入某些代码之前将代号为 4706 的警告屏蔽，之后又将其设为默认级别，首先要关注 4706 是否应该被屏蔽，还要关注如果将其设为默认是否与整个项目的设置有冲突。

应改为：

```
#pragma warning(push)
#pragma warning(disable:4706)
#include "somecode"
#pragma warning(pop)  // Compliant
```

改用这种方式之后不必再关注是否与整个项目的设置有冲突了。

相关

ID_warningDisabled

参考

SEI CERT MSC00-C

3.4 Comment

R3.4.1 关注 TODO、FIXME、XXX、BUG 等特殊注释

ID_specialComment precompile warning

TODO、FIXME、XXX、BUG 等特殊注释表示代码中存在问题，这种问题不应被遗忘，应有计划地予以解决。

当存在问题时在注释中及时记录是好的编程习惯，而且记录时最好有署名和日期。

示例：

```
void foo() {
    /*
     * Some plans...          // Bad, easy to forget
     */
}

void foo() {
    /* TODO:
     * Some plans...  --my name, date     // Good
     */
}
```

参考

CWE-546

R3.4.2 注释不可嵌套

ID_nestedComment　　precompile warning

嵌套的 /*...*/ 注释不符合标准，/* 与 */ 之间不应出现 /*，某些编译器可以接受嵌套，但不具备可移植性。

示例：

```
/*                          // #1
    /*                      // #2, Non-compliant
    nested comments
     */                     // #3
*/                          // #4, Non-compliant
```

根据标准，#1 处的 /* 与 #3 处的 */ 匹配，而 #4 处的 */ 处于失配状态。

依据

ISO/IEC 9899:1999 6.4.9(1)
ISO/IEC 9899:2011 6.4.9(1)

参考

MISRA C 2004 2.3
MISRA C 2012 3.1
MISRA C++ 2008 2-7-1

R3.4.3 注释应出现在合理的位置

ID_badCommentPosition　　precompile suggestion

注释应出现在段落的前后或行尾，不应出现在行首或中间，否则对阅读产生较大干扰，也可能产生标准未定义的问题。

示例：

```
#/*comment*/include "foo.h"          // Non-compliant
#include <bar.h /*comment*/>         // Non-compliant, undefined behavior

/*comment*/ int main()               // Non-compliant
{
    return a + b /*comment*/ + c;    // Non-compliant
}
```

应改为：

```
#include "foo.h"    // comment      // Compliant
#include <bar.h>    // comment      // Compliant

/*
 * comment                         // Compliant
 */
int main()
{
    return a + b + c;  // comment   // Compliant
}
```

例外：
当函数参数有默认值时，可以在函数实现时在参数声明的结尾用注释说明。

```
void foo(int i = 0);                // Declaration

void foo(int i /*= 0*/) {           // Let it go
}
```

3.5 Other

R3.5.1 除转义字符、宏定义之外不应使用反斜杠

ID_badBackslash　　precompile warning

反斜杠可用于标识转义字符，也可用于实现“伪换行”，即代码换行显示但在语法上并没有换行，一般用于宏定义，除此之外不应再使用反斜杠，否则没有实际意义，也会造成混乱。

示例：

```
#define M(x,y) if(x) {\   // Compliant
    foo(y);\              // Compliant
}

void foo() {
    if (condition1 \      // Non-compliant, meaningless
     || condition2) {
    }
}

int a\                    // Non-compliant, odd usage
b\
c = 123;

/\                        // Non-compliant, odd usage
/\ comment

void bar() {
    // comment  \         // Non-compliant, The next line is also commented out
    do_something();
}
```

4. Global

R4.1 全局名称应遵循合理的命名方式

ID_nameTooShort　　global suggestion

全局名称应具有标识性，长度不应过短，否则易与局部名称产生冲突。

本规则是 ID_badName 的特化。

示例：

```
// In global scope
const int i = 0;     // Non-compliant, name too short
typedef int t;       // Non-compliant, name too short
class A { .... };    // Non-compliant, name too short

int foo(int i) {
    return i + i;    // Confusing
}
```

名称适用的作用域范围越广，其长度也应该越长，建议全局名称长度不小于 3 个字符。

配置

minFunctionNameLength：全局函数名称长度下限，小于则报出
minNameSpaceNameLength：全局命名空间名称长度下限，小于则报出
minTypeNameLength：全局类型名称长度下限，小于则报出
minVariableNameLength：全局对象名称长度下限，小于则报出

相关

ID_badName

参考

C++ Core Guidelines NL.7

R4.2 为代码设定合理的命名空间

ID_missingNamespace　　global warning

命名空间是 C++ 项目的必要组成结构，可有效规避名称冲突等问题。

C++ 代码的顶层作用域应为具名非内联命名空间，命名空间名称应与项目名称相符，且具有标识性。

示例：

```
namespace NS {
    int foo();           // Compliant
    int foo(char*);      // Compliant
    int foo(wchar_t*);   // Compliant
}

int foo() {              // Non-compliant, it is not ‘int NS::foo()’
    ....
}

int NS::foo(char*) {     // Compliant
    ....
}

namespace NS {
    int foo(wchar_t*) {  // Compliant
        ....
    }
}
```

对于 main 函数和 extern "C" 声明的代码可不受本规则限制，如：

```
extern "C" int bar();  // Compliant

int main () {          // Compliant
    ....
}
```

相关

ID_usingNamespaceInHeader
ID_forbidUsingDirectives

参考

MISRA C++ 2008 7-3-1

R4.3 main 函数只应处于全局作用域中

ID_nonGlobalMain　　global warning

---

main 函数作为程序的入口，链接器需对其特殊处理，不应受命名空间等作用域的限制。

示例：

```
int main() {       // Compliant
    ....
}

namespace {
    int main() {   // Non-compliant
        ....
    }
}

namespace NS {
    int main() {   // Non-compliant
        ....
    }
}
```

参考

MISRA C++ 2008 7-3-2

R4.4 头文件中不应使用 using directive

ID_usingNamespaceInHeader　　global warning

在头文件的全局作用域中使用 using directive，极易造成命名冲突，且影响范围难以控制。

如果代码涉及多个命名空间，而这些命名空间中又有名称相同且功能相似的代码元素时，将造成难以排查的混乱。对于库的头文件，更应该严禁使用全局的 using directive，否则造成对用户命名空间的干扰。

示例：

```
// In a header file
namespace NS {
    void foo(short);
}

using namespace NS;   // Non-compliant
using namespace std;  // Non-compliant
```

下例展示的问题是头文件不同的包含顺序竟导致同一函数产生了不同的行为：

```
// In a.h
void foo(char);

namespace ns {
    void foo(int);
}

inline void bar() {
    foo(0);
}

// In b.h
namespace ns {}
using namespace ns;

// In a.cpp
#include "a.h"
#include "b.h"

void fun1() {
    bar();     // ‘bar’ calls ‘foo(char)’
}

// In b.cpp
#include "b.h"
#include "a.h"

void fun2() {
    bar();     // ‘bar’ calls ‘foo(int)’
}
```

头文件 a.h 和 b.h 以不同的顺序被包含，使 bar 函数调用了不同的 foo 函数，导致这种混乱的正是 b.h 中的 using directive。

相关

ID_forbidUsingDirectives

参考

C++ Core Guidelines SF.7
MISRA C++ 2008 7-3-6

R4.5 头文件中不应使用静态声明

ID_staticInHeader　　global warning

头文件中由 static 关键字声明的对象、数组或函数，会在每个包含该头文件的翻译单元或模块中生成副本造成数据冗余，如果将静态数据误用作全局数据也会造成逻辑错误。

类的静态成员不受本规则限制。

示例：

```
// In a header file
static int i = 0;   // Non-compliant

static int foo() {  // Non-compliant
    return i;
}
```

在编译每个包含该头文件的源文件时，变量 i 和函数 foo 都会生成不必要的副本。

在头文件中实现的内联或模版函数中，也不应使用静态声明，如：

```
// In a header file
inline void bar() {
    static MyType obj;  // Non-compliant
    ....
}
```

如果该头文件被不同的模块（so、dll、exe）包含，obj 对象会生成不同的副本，很可能造成逻辑错误。

另外， 由 const 或 constexpr 关键字修饰的常量也具有静态数据的特性，在头文件中定义常量也面对这种问题，基本类型的常量经过编译优化可以不占用存储空间（有取地址操作的除外），而对于非基本类型的常量对象或数组也不应在头文件中定义，建议采用单件模式，将其数据定义在 cpp 等源文件中，在头文件中定义访问这些数据的接口。

如：

```
// In myarr.h
using MyArr = int[256];
const MyArr& getMyArr();

// In myarr.cpp
#include "myarr.h"

const MyArr& getMyArr() {
    static MyArr arr = {
        1, 2, 3, ....
    };
    return arr;
}
```

在需要用到 arr 的地方，调用 getMyArr 函数，即可获取对该数组的引用，没有任何多余的数据产生，而且可保证在使用之前被有效初始化。

依据

ISO/IEC 14882:2011 3.5(3)

R4.6 头文件中不应定义匿名命名空间

ID_anonymousNamespaceInHeader　　global warning

头文件中定义了匿名命名空间，即相当于在头文件中定义了静态数据，头文件被多个源文件包含时便会造成数据冗余。

可参见 ID_staticInHeader 的进一步讨论。

示例：

```
// In a header file
namespace {  // Non-compliant
    void foo();
}
```

相关

ID_staticInHeader

依据

ISO/IEC 14882:2011 7.3.1.1

参考

C++ Core Guidelines SF.21
MISRA C++ 2008 7-3-3

R4.7 匿名命名空间中不应使用静态声明

ID_staticInAnonymousNamespace　　global warning

匿名命名空间中的元素已具有静态属性（internal linkage），不应再用 static 关键字修饰。

示例：

```
namespace {
    static int i = 0;   // Non-compliant
    static int foo() {  // Non-compliant
        return i++;
    }
}
```

例中 static 关键字是多余的。

应改为：

```
namespace {
    int i = 0;          // Compliant
    int foo() {         // Compliant
        return i++;
    }
}
```

依据

ISO/IEC 14882:2011 3.5(4)

R4.8 全局对象的初始化不可依赖未初始化的对象

ID_relyOnExternalObject　　global warning

全局对象的初始化不可依赖其他源文件中定义的对象，也不可依赖在其后面定义的对象。

示例：

```
extern int i;  // Defined in other translate unit
int j = i;     // Non-compliant
```

例中 i 是其他源文件中定义的对象，j 初始化时无法保证 i 已被正确初始化。不同源文件全局对象初始化的顺序在标准中是不确定的（indeterminately）。

又如：

```
extern int x;  // Defined after y
int y = x;     // Non-compliant
int x = 0;
```

在同一源文件中，x 在 y 的后面定义，语言标准规定了 x 的初始化也将在 y 后执行，而 y 依赖 x，所以 y 的初始化是无效的。

依据

ISO/IEC 14882:2011 3.6.2(2 3)
ISO/IEC 14882:2017 6.6.2(3)
ISO/IEC 14882:2017 6.6.3(2)

参考

C++ Core Guidelines I.22

R4.9 全局对象只应为常量或静态对象

ID_nonConstNonStaticGlobalObject　　global warning

非常量全局对象破坏了面向对象的封装理念，如果必须使用全局对象，应将其限定在文件范围之内。

本规则放宽了 ID_nonConstGlobalObj 的要求，对于 C++ 代码不建议选取本规则，对于 C 代码可酌情选取。

示例：

```
// In global scope
int i = 0;         // Non-compliant
static int j = 0;  // Let it go
const int k = 0;   // Compliant
```

相关

ID_nonConstGlobalObject

参考

C++ Core Guidelines I.2
C++ Core Guidelines CP.3
C++ Core Guidelines R.6

R4.10 全局对象只应为常量

ID_nonConstGlobalObject global warning

非常量全局对象与类的公有数据成员一样对外部的读写没有限制，破坏了面向对象的封装理念。

关于封装的讨论可参见 ID_nonPrivateData。

示例：

```
char foo;         // Non-compliant
extern char bar;  // Non-compliant, worse

void fun() {
    do_something(foo, bar);
}
```

改进方法（将全局变量和与其相关的函数封装成类）：

```
class A {
public:
    void fun() {
        do_something(foo, bar);
    }
private:
    char foo;  // Compliant
    char bar;  // Compliant
};
```

如果变量 foo、bar 确实具有全局意义，多个文件都需要访问，不妨将其单件化：

```
A& getObject() {
    static A a;
    return a;
}
```

用 getObject 函数获取对象，再由其成员函数对 foo、bar 进行读写，有效实现封装理念，而且可以保证对象在使用之前被有效初始化。

相关

ID_nonPrivateData

参考

C++ Core Guidelines I.2
C++ Core Guidelines CP.3
C++ Core Guidelines R.6

R4.11 全局对象不应同时被 static 和 const 关键字修饰

ID_staticAndConst　　global warning

由 const 关键字修饰的全局对象已具有静态属性（internal linkage），不应再用 static 关键字修饰。

示例：

```
static const int i = 123;  // Non-compliant, redundant ‘static’
```

应改为：

```
const int i = 123;         // Compliant
```

相关

ID_staticInHeader

依据

ISO/IEC 14882:2003 7.1.1(6)
ISO/IEC 14882:2011 7.1.1(7)

R4.12 全局或命名空间作用域中禁用 using directive

ID_forbidUsingDirectives　　⛔ global suggestion

将其他命名空间中的名称一并引入当前命名空间，是对命名空间机制的破坏。

标准库命名空间作为基础设施可被放过。

示例：

```
// In global namespace
using namespace std;          // Let it go
using namespace myspace0;     // Non-compliant
using namespace myspace1;     // Non-compliant

namespace myspace1 {
    using namespace myspace2  // Non-compliant
}
```

建议在函数作用域内用 using declaration 代替 using directive：

```
void foo() {
    using myspace2::Type;      // Compliant
    using myspace2::some_fun;  // Compliant

    Type x;
    some_fun(x);
}
```

相关

ID_usingNamespaceInHeader

参考

C++ Core Guidelines SF.6
MISRA C++ 2008 7-3-4

R4.13 避免无效的 using directive

ID_usingSelf　　🔥 global warning

用 using directive 引用当前命名空间属于无效代码，可能意味着某种错误。

示例：

```
namespace NS
{
    using namespace NS;  // Non-compliant, meaningless
}
```

R4.14 不应定义全局 inline 命名空间

ID_topInlineNamespace　　global suggestion

定义全局 inline 命名空间相当于没有命名空间，应在普通命名空间之内使用 inline 命令空间。

示例：

```
namespace V0 {
    int foo();
}

inline namespace V1 {  // Non-compliant
    int foo();
}
```

应该用普通命名空间加以限定：

```
namespace NS
{
    namespace V0 {
        int foo();
    }

    inline namespace V1 {  // Compliant
        int foo();
    }
}
```

R4.15 不可修改 std 命名空间

ID_stdNamespaceModified　　global warning

可以为用户定义的类型特化某些标准模板类，除此之外对 std 命名空间添加、修改甚至删除任何代码所导致的后果都是标准未定义的。

示例：

```
class MyType { .... };

namespace std
{
    size_t foo(const MyType& x);   // Non-compliant

    template <>
    struct hash<MyType> {
        size_t operator()(const MyType& x) const {
            return foo(x);
        }
    };
}
```

例中对 hash 标准模板类的特化是可被允许的，但在 std 命名空间中添加的 foo 函数是不被允许的。

应去掉 std 命名空间作用域声明，改为：

```
size_t foo(const MyType& x);   // OK

template <>
struct std::hash<MyType> {
    size_t operator()(const MyType& x) const {
        return foo(x);
    }
};
```

依据

ISO/IEC 14882:2011 17.6.4.2.1(1 2)
ISO/IEC 14882:2017 20.5.4.2.1(1 2)

参考

SEI CERT DCL58-CPP

5. Type

5.1 Class

R5.1.1 类的非常量数据成员均应为 private

ID_nonPrivateData　　type suggestion

类的数据成员均应设为私有，对外统一由成员函数提供访问方法。

将类的所有接口都实现为成员函数，由成员函数按指定逻辑读写数据，以便保证有效地改变对象状态。良好的接口设计会对代码的职责进行合理划分，显著提升可维护性。理想状态下，当有错误需要修正或有功能需要调整时，只改动相关接口的实现即可，调用接口的代码不需要改动，从而将改动降到最低。这种设计的基础便是将数据设为私有，只能由本类的成员函数访问，否则数据可被各个模块随意读写，当有一处需要改动时，很难控制其影响范围。

示例：

```
struct Fraction {
    int n, d;  // Bad

    double result() {
        return double(n) / d;
    }
};
```

例中成员 n 和 d 可被外部随意访问，如果 d 被设为 0 则无法进行除法运算，破坏了对象的有效性。

对数据的限定以及数据之间的内在关系应由成员函数统一维护，不暴露给类的使用者，这便是面向对象的封装理念，也是 C++ 语言的核心理念之一。

应改为：

```
class Fraction {
public:
    Fraction(int x, int y): n(x), d(y) {
        if (!d) {
            throw BadFraction();
        }
    }

    double result() noexcept {
        return double(n) / d;
    }

private:
    int n, d;  // Good
};
```

例外：
常量数据成员不可被改变，所以可不受本规则约束。

相关

ID_protectedData
ID_mixPublicPrivateData

参考

MISRA C++ 2008 11-0-1

R5.1.2 类的非常量数据成员不应定义为 protected

ID_protectedData　　type suggestion

---

protected 数据成员在派生类中仍可随意读写，破坏了封装理念。

本规则是 ID_nonPrivateData 的特化，关于封装的进一步讨论可参见 ID_nonPrivateData。

示例：

```
class A {
    ....
protected:
    int member;  // Non-compliant
};
```

应改为由接口访问：

```
class A {
    ....
protected:
    int access_member();  // Compliant
};
```

例外：
常量数据成员不可被改变，所以可不受本规则约束。

相关

ID_mixPublicPrivateData
ID_nonPrivateData

参考

C++ Core Guidelines C.9
C++ Core Guidelines C.133

R5.1.3 类不应既有 public 数据成员又有 private 数据成员

ID_mixPublicPrivateData　　type suggestion

类的设计应遵循：

- 成员之间没有依赖关系，且都可以随意被读写时，则都应声明为 public
- 成员之间有依赖关系，或成员的状态会影响到整个对象的状态时，则都应声明为 private

否则应对类进行改造或拆分。

面向对象的封装理念更倾向于将所有数据成员都设为 private，由成员函数按指定逻辑控制每个成员的读写方法，以供外部访问，对代码的职责进行有效的划分，从而提高可维护性并降低风险，关于封装的进一步讨论可参见 ID_nonPrivateData。

示例：

```
class A {  // Non-compliant
public:
    int n;
    ....
private:
    int d;
};
```

应改为：

```
class A {  // Compliant
public:
    int method_for_n();
    ....
private:
    int n, d;
};
```

例外：
常量数据成员不可被改变，所以可不受本规则约束。

相关

ID_nonPrivateData
ID_protectedData

参考

C++ Core Guidelines C.9
C++ Core Guidelines C.134

R5.1.4 有虚函数的基类应具有虚析构函数

ID_missingVirtualDestructor type warning

为了避免意料之外的资源泄漏，有虚函数的基类，都应该具有虚析构函数。

当通过基类指针析构派生类对象时，如果基类没有虚析构函数，派生类对象的析构函数是无法被执行的，造成不易排查的资源泄漏。

示例：

```
class A {
public:
    A() {}
   ~A() {}  // Non-compliant, missing ‘virtual’
    virtual size_t size() = 0;
};

class B: public A {
    int* x;
    size_t n;

public:
    B(size_t s): n(s), x(new int[s]) {
    }

   ~B() {
        delete[] x;
    }

    size_t size() override {
        return n;
    }
};
```

按下列调用：

```
A* p = new B(32);
cout << p->size();  // OK, output 32
delete p;           // But only ‘A::~A’ is called, ‘B::x’ leaks
```

由于 A 的析构函数不是虚函数，所以 delete p 只调用了 A 的析构函数，导致派生类对象中的资源没有得到释放。

依据

ISO/IEC 14882:2003 12.4(7) 5.3.5(3)
ISO/IEC 14882:2011 12.4(9) 5.3.5(3)
ISO/IEC 14882:2017 15.4(10) 8.3.5(3)

参考

CWE-1079
CWE-1087
CWE-1045
C++ Core Guidelines C.35
C++ Core Guidelines C.127

R5.1.5 用虚基类避免冗余的基类实例

ID_diamondInheritance　　type suggestion

当一个类有多个基类，这些基类又继承自同一个类时，会产生多个不同的基类实例，造成逻辑上的冗余和不必要的存储开销。

示例：

```
struct A {
    int i = 0;
};

class B: public A {};
class C: public A {};
class D: public B, public C {};

void foo(D& d) {
    d.i = 1;     // Complie error
    d.B::i = 1;  // Odd
    d.C::i = 1;  // Odd
}
```

在 D 类对象 d 中，基类 A 的成员 i 有两个不同的实例，d 不能直接访问 i，只能通过 d.B::i 或 d.C::i 这种怪异的方式访问。

将共同的基类设为虚基类可以解决这种问题：

```
class B: virtual public A {};
class C: virtual public A {};
class D: public B, public C {};

void foo(D& d) {
    d.i = 1;     // OK
}
```

注意，直接将虚基类指针转为派生类指针是标准未定义的行为，如：

```
void bar(A* a) {
    B* p = (B*)a;  // Undefined behavior
    ....
}
```

这种情况一般不会通过编译，但在较低版本的编译器中也有例外。

应改用 dynamic_cast：

```
B* p = dynamic_cast<B*>(a);  // OK
```

依据

ISO/IEC 14882:2011 10.1(4 5 6 7)
ISO/IEC 14882:2003 5.2.9(5 8)-undefined
ISO/IEC 14882:2011 5.2.9(11 12)-undefined

参考

C++ Core Guidelines C.137

R5.1.6 存在赋值运算符或析构函数时，不应缺少拷贝构造函数

ID_missingCopyConstructor　　type warning

三个紧密相关的函数：

1. 拷贝构造函数
2. 析构函数
3. 赋值运算符

当这三个函数中的任何一个函数被定义时，说明对象在复制和资源管理方面有特定的行为，所以其他两个函数也需要被定义，这种规则称为“Rule of three”。如果缺少某个函数，编译器会生成相关函数，但其特定需求不会被实现。

示例：

```
class A {    // Non-compliant
    int* p;

public:
    A(): p(new int[123]) {
    }

   ~A() {
        delete[] p;
    }
};
```

例中 A 有析构函数，但没有拷贝构造函数和赋值运算符，编译器会生成相关默认函数，但只进行变量值的复制，使多个对象的成员 p 指向同一块内存区域，导致析构时内存被重复释放，所以应定义拷贝构造函数和赋值运算符重新分配内存并复制数据。

注意，当类只负责成员的组合而没有特殊的复制或析构需求时，这三个函数就都不要定义，这种规则称为“Rule of zero”。

示例：

```
class B {
    string a, b;

public:
    B(const B& rhs): a(rhs.a), b(rhs.b) {  // Unnecessary
    }

    B& operator = (const B& rhs) {  // Unnecessary
        a = rhs.a;
        b = rhs.b;
    }

   ~B() {  // Unnecessary
    }
};
```

例中 B 只涉及字符串对象的组合，复制和析构可交由成员对象完成，其拷贝构造、赋值运算符及析构函数是不必要的，应该去掉，编译器会进行更好的处理。

同理，在遵循 C++11 及之后标准的代码中，对于：

1. 拷贝构造函数
2. 析构函数
3. 赋值运算符
4. 移动拷贝构造函数
5. 移动赋值运算符

当定义了这五个函数中的任何一个函数时，其他四个函数也需要定义，这种规则称为“Rule of five”。

如果确实不需要某个函数，也需要用“=delete”指明，以明确约束对象的行为。

相关

ID_missingDestructor
ID_missingCopyAssignOperator

参考

C++ Core Guidelines C.21

R5.1.7 存在拷贝构造函数或析构函数时，不应缺少拷贝赋值运算符

ID_missingCopyAssignOperator　　type warning

三个紧密相关的函数：

1. 拷贝构造函数
2. 析构函数
3. 赋值运算符

当这三个函数中的任何一个函数被定义时，其他两个函数也需要被定义，详见“Rule of three”。

值得强调的是，如果确实不需要赋值运算符，需明确将其声明为“=delete”，如果确实只需要浅拷贝，需将其声明为“=default”，这样明确了复制对象时的行为，规避意料之外的错误。

示例：

```
class A {  // Non-compliant, missing assignment operator
public:
    A();
    A(const A&);
   ~A();
};
```

应明确定义赋值运算符：

```
class A {  // Compliant
public:
    A();
    A(const A&);
   ~A();

    A& operator = (const A&);  // Assignment operator
};
```

相关

ID_missingDestructor
ID_missingCopyConstructor

参考

C++ Core Guidelines C.21

R5.1.8 存在拷贝构造函数或赋值运算符时，不应缺少析构函数

ID_missingDestructor type warning

三个紧密相关的函数：

1. 拷贝构造函数
2. 析构函数
3. 赋值运算符

当这三个函数中的任何一个函数被定义时，其他两个函数也需要被定义，详见“Rule of three”。

示例：

```
class A {  // Non-compliant, missing destructor
public:
    A();
    A(const A&);
    A& operator = (const A&);
};
```

应明确定义析构函数：

```
class A {  // Compliant
public:
    A();
    A(const A&);
    A& operator = (const A&);

   ~A();   // Destructor
};
```

相关

ID_missingCopyConstructor
ID_missingCopyAssignOperator

参考

C++ Core Guidelines C.21
C++ Core Guidelines C.30
C++ Core Guidelines C.33

R5.1.9 存在移动构造函数时，不应缺少移动赋值运算符

ID_missingMoveAssignOperator type warning

五个紧密相关的函数：

1. 拷贝构造函数
2. 析构函数
3. 赋值运算符
4. 移动拷贝构造函数
5. 移动赋值运算符

当这五个函数中的任何一个函数被定义时，其他四个函数也需要被定义，详见“Rule of five”，尤其是存在移动构造函数时，不应缺少移动赋值运算符。

相关

ID_missingCopyAssignOperator

参考

C++ Core Guidelines C.21

R5.1.10 存在移动赋值运算符时，不应缺少移动构造函数

ID_missingMoveConstructor type warning

五个紧密相关的函数：

1. 拷贝构造函数
2. 析构函数
3. 赋值运算符
4. 移动拷贝构造函数
5. 移动赋值运算符

当这五个函数中的任何一个函数被定义时，其他四个函数也需要被定义，详见“Rule of five”，尤其是存在移动赋值运算符时，不应缺少移动构造函数。

相关

ID_missingCopyConstructor

参考

C++ Core Guidelines C.21

R5.1.11 可接受一个参数的构造函数需用 explicit 关键字限定

ID_missingExplicitConstructor　　type suggestion

为了避免意料之外的类型转换，可接受一个参数的构造函数应该用 explicit 关键字限定。

示例：

```
class String {
public:
    String(int capacity);   // Missing ‘explicit’
    ....
};

void foo(const String&);

int bar() {
    foo(100);  // Can be compiled, but very odd
}
```

由于 String 类的构造函数接受一个 int 型参数，foo(100) 相当于将 100 隐式转为 String 类的对象，这种隐式转换是怪异的，也往往意味着意料之外的错误。

应改为：

```
class String {
public:
    explicit String(int capacity);  // OK
    ....
};
```

这样 foo(100) 这种写法便不会通过编译。

例外：
对于拷贝、移动构造函数不受本规则约束，如果将拷贝、移动构造函数声明为 explicit 则无法再按值传递参数或按值返回对象。

```
class String {
public:
    String(const String&);  // Explicit or not depends on your design intent
    String(String&&);        // ditto
    ....
};
```

在类的接口设计中，应尽量减少隐式转换以避免不易察觉的问题。

相关

ID_missingExplicitConvertor

参考

C++ Core Guidelines C.46
MISRA C++ 2008 12-1-3

R5.1.12 重载的类型转换运算符需用 explicit 关键字限定

ID_missingExplicitConvertor　　type suggestion

为了避免意料之外的类型转换，重载的类型转换运算符需用 explicit 关键字限定。

示例（选自 C++ Core Guidelines）：

```
struct S1 {
  string s;
  // ...
  operator char*() { return s.data(); }  // BAD, likely to cause surprises
};

struct S2 {
  string s;
  // ...
  explicit operator char*() { return s.data(); }
};

void f(S1 s1, S2 s2)
{
  char* x1 = s1;  // OK, but can cause surprises in many contexts
  char* x2 = s2;  // error (and that's usually a good thing)
  char* x3 = static_cast<char*>(s2); // we can be explicit (on your head be it)
}
```

一个由隐式类型转换引发的错误：

```
S1 ff();

char* g()
{
  return ff();
}
```

S1 的类型转换函数被隐式调用，然而当 g 函数返回后，临时对象被销毁，返回的指针为无效地址。

在类的接口设计中，应尽量减少隐式转换以避免不易察觉的问题。

相关

ID_missingExplicitConstructor

参考

C++ Core Guidelines C.164

R5.1.13 不应过度使用 explicit 关键字

ID_excessiveExplicit type warning

对类的拷贝、移动以及不接受 1 个参数的构造函数一般不用 explicit 限定，否则有损代码的易用性和可扩展性。

示例：

```
class A {
public:
    explicit A(const A&);  // In general, 'explicit' is not required
    explicit A(A&&);       // Ditto
    explicit A(int, int);  // Ditto
    ....
};
```

当类的拷贝、移动构造函数被 explicit 限定时，无法再按值传递参数或按值返回对象，当不接受 1 个参数的构造函数被 explicit 限定时，无法再用初始化列表定义临时对象，如下代码将无法通过编译：

```
void foo(A);
void bar(const A&);

A a(1, 2);

foo(a);       // Compile error
bar({3, 4});  // Compile error
```

参考

C++ Core Guidelines C.46

R5.1.14 带模板的赋值运算符不应覆盖拷贝或移动赋值运算符

ID_roughTemplateAssignOperator type warning

带模板的赋值运算符覆盖拷贝或移动赋值运算符，很可能导致意料之外的错误。

示例：

```
class A {
public:
    template <class T> A& operator = (T x);  // Non-compliant
    ....
};
```

例中的赋值运算符可以同时作为普通赋值运算符、拷贝赋值运算符和移动赋值运算符，是一种混乱的设计。

应明确其拷贝赋和移动赋值运算符：

```
class A {
public:
    A& operator = (const A&);
    A& operator = (A&&);
    template <class T> A& operator = (T x);  // Compliant
    ....
};
```

相关

ID_roughTemplateConstructor

参考

MISRA C++ 2008 14-5-3

R5.1.15 带模板的构造函数不应覆盖拷贝或移动构造函数

ID_roughTemplateConstructor type warning

带模板的构造函数覆盖拷贝或移动构造函数，很可能导致意料之外的错误。

示例：

```
class A {
public:
    template <class T> A(T x);  // Non-compliant
    ....
};
```

例中的构造函数可以同时作为普通构造函数、拷贝构造函数和移动构造函数，是一种混乱的设计。

应明确其拷贝和移动构造函数：

```
class A {
public:
    A(const A&);
    A(A&&);
    template <class T> A(T x);  // Compliant
    ....
};
```

相关

ID_roughTemplateAssignOperator

参考

MISRA C++ 2008 14-5-2

R5.1.16 抽象类禁用拷贝赋值运算符

ID_unsuitableCopyAssignOperator　　⛔ type warning

抽象类没有独立的对象，不应存在拷贝赋值运算符，否则赋值是不完整的。

示例：

```
struct A {
    virtual ~A() = 0;
    A& operator = (const A&);  // Non-compliant
    ....
};

void foo(A& x, A& y) {
    x = y;  // Incomplete assignment
}
```

例中 foo 函数的参数只能是 A 的派生类对象，派生类对象调用基类的赋值运算符会造成数据不一致等问题。

应将赋值运算符设为 =delete 或 private：

```
struct A {
    virtual ~A() = 0;
    A& operator = (const A&) = delete;  // Compliant
    ....
};
```

使这种问题无法通过编译，可及时更换正确的方法。

参考

MISRA C++ 2008 12-8-2
C++ Core Guidelines C.67

R5.1.17 数据成员的数量应在规定范围之内

ID_tooManyFields　　type warning

类或联合体的数据成员过多意味着一个逻辑或功能单位承担了过多的职责，违反了模块化设计理念，是难以维护的。

示例：

```
class C
{
    // ... 3000 members ...
    // Who has the courage to read?
};

union U
{
    // ... 3000 members ...
    // It's actually the hell...
};
```

配置

maxClassFieldsCount：类数据成员的数量上限，超过则报出
maxUnionFieldsCount：联合体数据成员的数量上限，超过则报出

R5.1.18 存在构造、析构或虚函数的类不应采用 struct 关键字

ID_unsuitableStructTag　　type suggestion

为了便于区分简单结构体和具有封装或多态属性的类，建议 struct 关键字只用于结构体，其他情况均采用 class 关键字。

示例：

```
struct A {     // Non-compliant
    int x, y;

    A();
   ~A();
};

struct B {     // Compliant
    int x, y;
};

class C {      // Compliant
    int x, y;

public:
    C();
   ~C();
};
```

参考

C++ Core Guidelines C.2
C++ Core Guidelines C.8

5.2 Enum

R5.2.1 同类枚举项的值不应相同

ID_duplicateEnumerator　　type warning

枚举项用于标记不同的事物，名称不同但值相同的枚举项往往意味着错误。

示例：

```
enum Color {
    red = 1,
    yellow = 2,
    blue = 2,    // Non-compliant, see ‘yellow’
};
```

例中三个枚举项应分别表示三种颜色，但 blue 与 yellow 的值相同会造成逻辑错误。

又如：

```
enum Fruit {
    apple,
    pear,
    grape,
    favourite = grape,  // Non-compliant
};
```

例中 Fruit 定义了三种水果，而 favourite 表示最喜欢的水果，与其他枚举项不是同一层面的概念，不应聚为一类。

应采用更结构化的方式：

```
enum Fruit {
    apple, pear, grape
};

Fruit favourite() {
    return grape;
}
```

参考

C++ Core Guidelines Enum.8

R5.2.2 合理初始化各枚举项

ID_casualInitialization　　type suggestion

合理初始化各枚举项，只应从下列方式中选择一种：

- 全不初始化
- 只初始化第一个
- 全部初始化为不同的值

示例：

```
enum Colour {
    red,
    blue,
    green,
    yellow = 2   // Non-compliant
};
```

应改为：

```
enum Colour {
    red,
    blue,
    green,
    yellow      // Compliant
};
```

相关

ID_duplicateEnumerator

参考

MISRA C 2004 9.3
MISRA C++ 2008 8-5-3

R5.2.3 不应使用匿名枚举声明

ID_forbidUnnamedEnum　　⛔ type suggestion

匿名枚举声明相当于在当前作用域定义常量，但类型不够明确。

示例：

```
enum { rabbit = 0xAA, carrot = 1234 };  // Non-compliant
```

如果无法确定枚举类型的名称，也意味着各枚举项不应聚为一类。

应改为：

```
const int rabbit = 0xAA;  // Compliant
const int carrot = 1234;  // Compliant
```

参考

C++ Core Guidelines Enum.6

R5.2.4 用 enum class 取代 enum

ID_forbidUnscopedEnum ⛔ type suggestion

传统 C 枚举没有有效的类型和作用域控制，极易造成类型混淆和名称冲突，在 C++ 语言中建议改用 enum class。

示例：

```
enum E {     // Non-compliant
    e0 = 0,
    e1 = 1,
    e2 = -1
};

E foo();

void bar() {
    if (foo()) {  // ‘e1’ or ‘e2’??
        ....
    }
}
```

传统 C 枚举值与 int 等类型可以随意转换，如果 e0 和 e2 表示某种错误情况，e1 表示正确情况，那么 bar 函数中对 foo 返回值的判断就是错误的，这也是一种常见问题，C++11 提出了 enum class 的概念加强了类型检查，提倡在新项目中尽量使用 enum class。

应改为：

```
enum class E {  // Compliant
    e0 = 0,
    e1 = 1,
    e2 = -1
};

void bar() {
    if (foo() == E::e1) {  // OK
        ....
    }
    if (foo()) {  // Compile error, cannot cast the enum class casually
        ....
    }
}
```

依据

ISO/IEC 14882:2011 7.2(2)

参考

C++ Core Guidelines Enum.3

5.3 Union

R5.3.1 联合体内禁用非基本类型的对象

ID_forbidNonBasicField　　⛔ type warning

因为联合体成员之间共享内存地址，所以成员具有构造或析构函数时会导致混乱。

C++98/03 禁止具有拷贝构造函数或析构函数的对象出现在联合体中，C++11 解除了这条禁令，但在语言层面上不保障正确性，相当于把问题抛给了用户。

示例：

```
union U {
    int i;
    string s;  // Non-compliant

    U(int x): i(x) {
    }

    U(const char* x) {
        new(&s) string(x);
    }

   ~U() {
        s.~string();
    }
};

U u(1);
u.s = "abc";  // No error, no warning, just crash
```

示例代码在某些环境中会崩溃，原因是没能正确区分对象当前持有的类型，执行了错误的构造或析构过程。

正确的做法是在类中用一个成员变量记录当前持有的类型，再将匿名 union 和类的构造函数以及析构函数相关联，从而根据当前持有的类型正确地初始化或销毁对象。

依据

ISO/IEC 14882:1998 9.5(1)
ISO/IEC 14882:2003 9.5(1)
ISO/IEC 14882:2011 9.5(2 3 4)

R5.3.2 禁用在类之外定义的联合体

ID_forbidNakedUnion ⛔ type suggestion

联合体各成员共享存储地址，易引发意料之外的错误。如果一定要使用联合体，需对其进行一定的封装，避免对成员的错误访问。

不应出现：

- 在命名空间作用域内定义的联合体
- 在类中定义的具有 public 访问权限的联合体

示例：

```
union U {   // Non-compliant, global union
    ....
};

class A {
public:
    union {   // Non-compliant, public union
        ....
    };
};

class B {
public:
    // ... Interfaces about the union
private:
    union {   // Compliant, the union is under control
        ....
    };
};
```

类的公有数据成员本来就违反了封装原则，如果这种公有数据成员又在联合体中，就进一步加大了风险。

相关

ID_forbidUnion

参考

C++ Core Guidelines C.181
MISRA C 2004 18.4
MISRA C 2012 19.2
MISRA C++ 2008 9-5-1

R5.3.3 禁用联合体

ID_forbidUnion　　⛔ type suggestion

联合体的问题主要有：

- 无法只通过对象获取当前有效的成员
- 访问不同的成员相当于不安全的类型转换
- 对非基本类型的成员造成构造和析构的混乱
- 不能作为基类

这些问题在本质上是对类型理念的破坏，面向对象的程序设计应避免使用联合体。

示例：

```
union U {  // Non-compliant
    int i;
    char c;
};

U u;
u.i = 1000;
cout << u.c << '\n';  // Equivalent to a cast without any restrictions
```

对联合体的使用也相当于一种没有限制的强制类型转换，在 C++ 中建议用 std::variant 或 std::any 取代联合体：

```
std::variant<int, char> u;
u = 123;
cout << get<int>(u) << '\n';   // OK
cout << get<char>(u) << '\n';  // Throw ‘std::bad_variant_access’
```

std::variant 可以有效记录对象当前持有的类型，如果以不正确的类型访问对象会及时抛出异常。

本规则比 ID_forbidNakedUnion 更严格，针对所有联合体。

相关

ID_forbidNakedUnion

参考

MISRA C 2004 18.4
MISRA C 2012 19.2
MISRA C++ 2008 9-5-1

6. Declaration

6.1 Naming

R6.1.1 遵循合理的命名方式

ID_badName　　declaration suggestion

应遵循易于读写，并可准确表达代码意图的命名方式。

不应出现下列情况：

- 超长的名称
- 易造成混淆或冲突的名称
- 无意义或意义过于空泛的名称
- 不易于读写的名称
- 有违公序良俗的名称

示例：

```
namespace xxx  // Bad, meaningless name
{
    void fun(int);  // Bad, vague

    const int nVarietyisthespiceoflife = 123;  // Bad, hard to read
}
```

例中 xxx、fun 这种无意义或过于空泛的名称是不符合要求的，名称中各单词间应有下划线或大小写变化，否则是不便于读写的。本文示例中出现的 foo、bar 等名称，意在代指一般的代码元素，仅作示例，实际代码中不应出现。

不良命名方式甚至会导致标准未定义的行为，如：

```
extern int identifier_of_a_very_very_long_name_1;
extern int identifier_of_a_very_very_long_name_2;   // Dangerous
```

注意，如果两个名称有相同的前缀，而且相同前缀超过一定长度时是危险的，有可能导致编译器无法有效区分相关名称，造成标准未定义的问题。C 语言标准指明，保证名称前 31 位不同即可避免这种问题，可参见 ISO/IEC 9899:2011 5.2.4.1 的相关规定。

不建议采用相同“长前缀”+ 不同“短后缀”的命名方式，这种名称非常容易形成笔误或由复制粘贴造成错误，如：

```
struct BinExpr {
    BinExpr* sub0;   // Bad
    BinExpr* sub1;   // Bad
};
```

设 BinExpr 是“二元表达式”类，sub0、sub1 为左右子表达式，这种命名方式应改进：

```
struct BinExpr {
    BinExpr* left;   // Better
    BinExpr* right;  // Better
};
```

配置

maxWordLength：连续无大小写变化的字符个数上限，超过则报出

依据

ISO/IEC 9899:1999 5.2.4.1(1)
ISO/IEC 9899:1999 6.4.2.1(6)-undefined
ISO/IEC 9899:2011 5.2.4.1(1)
ISO/IEC 9899:2011 6.4.2.1(6)-undefined

参考

C++ Core Guidelines NL.19
C++ Core Guidelines ES.8
MISRA C 2004 5.1
MISRA C 2012 5.1

R6.1.2 不应定义具有保留意义的名称

ID_reservedName　　declaration suggestion

自定义的名称不应与关键字、标准库或系统中的名称重复，否则极易造成阅读和维护上的困扰。

对于宏，本规则特化为 ID_macro_defineReserved、ID_macro_undefReserved，如果宏名称出现这种问题，会导致标准未定义的错误。

示例：

```
#include <errno.h>

struct A {
    void foo() {
        if (errno != 0) {  // Which errno?
            ....
        }
    }
private:
    int errno;  // Non-compliant
};
```

例中成员变量 errno 与标准库中的 errno 名称相同，不便于区分是自定义的还是系统定义的，造成不必要的困扰。

为避免冲突和误解，以下命名方式可供参考：

- 避免名称以下划线开头
- 无命名空间限制的全局名称以模块名称开头

- 从名称上体现作用域，如全局对象名以 g_ 开头，成员对象名以 m_ 开头或以 _ 结尾
- 从名称上体现类别，如宏名采用全大写字母，类型名以大写字母开头，函数或对象名以小写字母开头

本规则集合对具体的命名方式暂不作量化要求，但代码编写者应具备相关意识。

相关

ID_macro_defineReserved
ID_macro_undefReserved

依据

ISO/IEC 9899:2011 7.1.3(1)

参考

SEI CERT DCL37-C
SEI CERT DCL51-CPP
MISRA C 2012 21.2
MISRA C++ 2008 17-0-1
MISRA C++ 2008 17-0-2
MISRA C++ 2008 17-0-3

R6.1.3 局部名称不应被覆盖

ID_hideLocal declaration warning

嵌套的作用域中不应出现相同的名称，否则干扰阅读，极易产生误解。

示例：

```
int foo() {
    int i = 0;  // Declares an object ‘i’
    if (cond) {
        int i = 1;  // Non-compliant, hides previous ‘i’
        ....
    }
    return i;
}
```

在一个函数中出现了多个名为 i 的变量，当实际代码较为复杂时，很容易出现意图与实现不符的问题。

参考

CWE-1109
C++ Core Guidelines ES.12
MISRA C 2004 5.2
MISRA C 2012 5.3
MISRA C++ 2008 2-10-2

R6.1.4 成员名称不应被覆盖

ID_hideMember declaration warning

如果成员函数内的局部名称与成员名称相同，会干扰阅读，易产生误解。

示例：

```
class A {
    int i = 0;          // Member object ‘i’

public:
    int foo() {
        int i = 0;      // Non-compliant, hides the member ‘i’
        return bar(i);
    }

    int bar(int i) {    // Non-compliant, hides the member ‘i’
        return i + i;   // Which ‘i’ is used?
    }
};
```

建议成员变量遵循统一的命名约定，如以“_”结尾或以“m_”开头，可有效规避这类问题：

```
class A {
    int i_ = 0;         // Member object ‘i_’

public:
    int foo() {
        int i = 0;      // Compliant
        return bar(i);
    }

    int bar(int i) {    // Compliant
        return i_ + i;
    }
};
```

参考

CWE-1109
MISRA C 2004 5.2
MISRA C 2012 5.3
MISRA C++ 2008 2-10-2

R6.1.5 类型名称不应与对象或函数名称相同

ID_duplicatedName　　declaration suggestion

如果不同的代码元素使用相同的名称，极易造成困扰。

示例：

```
struct A {
    ....
};

enum {
    A, B, C    // Non-compliant
};

size_t x = sizeof(A);  // What is ‘A’?
```

例中类名 A 与枚举项 A 重名，sizeof(A) 的意义是非常令人困惑的。

参考

MISRA C++ 2008 2-10-6

R6.1.6 不应存在拼写错误

ID_misspelling　　declaration suggestion

代码中不应存在拼写错误，尤其是供他人调用的代码，如命名空间名称、类的公有成员名称，全局函数名称等，更不应存在拼写错误。

错误拼写会使代码的使用者对代码的质量产生疑虑，而且这种代码被大量引用后也不便于改正。

示例：

```
class A {
public:
    virtual void destory() = 0;  // Non-compliant, should be ‘destroy’
};
```

例中“destory”函数的名称有拼写错误，应改为“destroy”。

6.2 Qualifier

R6.2.1 const、volatile 不应重复

ID_qualifierRepeated　　declaration error

重复的 const 或 volatile 限定符是没意义的，也可能意味着某种错误。

示例：

```
const const char* p0 = "....";   // Non-compliant
const char const* p1 = "....";   // Non-compliant
char* const const p2 = "....";   // Non-compliant
```

对于 p0 和 p1，const 重复修饰 char，很可能应该修饰 * 号，属于常见笔误，应改为：

```
const char * const p0 = "....";  // Compliant
const char * const p1 = "....";  // Compliant
```

对于 p2，const 重复修饰 * 号，符合语言文法，但没有实际意义，很可能应该修饰 char，应改为：

```
const char * const p2 = "....";  // Compliant
```

相关

ID_badQualifierPosition

R6.2.2 const、volatile 修饰指针类型的别名是可疑的

ID_qualifierForPtrAlias　　declaration suspicious

如果 const、volatile 修饰指针类型的别名，很可能会造成意料之外的问题。

示例：

```
struct Type {
    void foo();
    void foo() const;
};

typedef Type* Alias;

void bar(const Alias a) {  // Rather suspicious
    a->foo();              // Calls ‘void Type::foo();’
}
```

例中 Alias 是 Type* 的别名，“const Alias a”很容易引起误解，好像对象是不可被改变的，但实际上 a 的类型是 Type *const，const 限定的是指针，而不是指针指向的对象，这种情况下，对象仍可以被修改，其调用的函数也可能与预期不符。

应避免为指针类型定义别名，如果必须定义应提供常量和非常量两种别名，如：

```
typedef Type* Alias;
typedef const Type* ConstAlias;

void bar(ConstAlias a) {
    a->foo();              // Calls ‘void Type::foo() const;’
}
```

注意，如果 const、volatile 修饰引用的别名则是错误的，详见 ID_qualifierInvalid。

相关

ID_qualifierInvalid

参考

SEI CERT DCL05-C

R6.2.3 const、volatile 不可修饰引用

ID_qualifierInvalid　　declaration error

C++ 标准规定，const 或 volatile 可修饰指针，但不可修饰引用，否则起不到任何作用。

示例：

```
int a = 0;
int &const i = a;     // Non-compliant
int &volatile j = a;  // Non-compliant
```

修饰 & 号的 const 和 volatile 是无效的，i 可被随意修改，j 也可能被优化。

应去掉限定符，或使限定符修饰引用的对象：

```
const int& i = a;     // Compliant
volatile int& j = a;  // Compliant
```

注意，如果限定符修饰引用类型的别名，会引起很大误解，如：

```
typedef int& int_r;   // Reference type alias, bad
const int_r r0 = a;   // Non-compliant, r0 is not a const-reference at all
const int_r& r1 = a;  // Non-compliant, r1 is not a const-reference at all
```

例中 r0 像是一个常量对象，而 r1 像是常量对象的引用，但 const int_r 展开后相当于 int & const，r0 不是常量，r1 也不是常量的引用。

依据

ISO/IEC 14882:2003 8.3.2(1)
ISO/IEC 14882:2011 8.3.2(1)
ISO/IEC 14882:2017 11.3.2(1)

R6.2.4 const、volatile 限定类型时应出现在左侧

ID_badQualifierPosition　　declaration suggestion

语言允许 const、volatile 出现在类型名的左侧，也可以出现在其右侧，甚至可以出现在基本类型名的中间，为了提高可读性，应对其位置进行统一规范。

示例：

```
unsigned int const a = 0;          // Non-compliant
long double const volatile b = 0;  // Non-compliant
```

应改为：

```
const unsigned int a = 0;          // Compliant
const volatile long double b = 0;  // Compliant
```

const、volatile 出现在类型名右侧时，易与 * 号造成误解，如：

```
const char const* p = "....";      // Non-compliant
```

应改为：

```
const char * const p = "....";     // Compliant
```

相关

ID_sandwichedModifier
ID_badSpecifierPosition

参考

C++ Core Guidelines NL.26

R6.2.5 const、volatile 等关键字不应出现在基本类型名称的中间

ID_sandwichedModifier　　declaration suggestion

某些基本类型可由多个符号组成，const 或 volatile 等关键字不应出现在这些符号的中间，否则可读性较差。

示例：

```
const long volatile long cvll = 0;    // Non-compliant
long const double volatile cvld = 0;  // Non-compliant
```

应改为：

```
const volatile long long cvll = 0;    // Compliant
const volatile long double cvld = 0;  // Compliant
```

本规则对下列关键字有同样的要求：

```
const、volatile、
inline、virtual、explicit、
register、static、thread_local、extern、mutable、
friend、typedef、constexpr
```

相关

ID_badQualifierPosition
ID_badSpecifierPosition

参考

C++ Core Guidelines NL.26

R6.2.6 避免用常量字符串对非常量字符串指针赋值

ID_missingConst declaration warning

用常量字符串对非常量字符串指针赋值，相关内存被修改会导致标准未定义的问题。

示例：

```
char* p = "a string literal";  // Non-compliant
....
p[x] = '\0';   // Undefined behaivor
```

例中 p 指向常量字符串，通过 p 修改相关内存区域会种程序产生未定义的行为。

应改为：

```
const char* p = "a string literal";  // Compliant
....
p[x] = '\0';   // Compile-time protected
```

改为常量字符串指针后，错误的操作无法通过编译。

依据

ISO/IEC 14882:2003 2.13.4(2)-undefined
ISO/IEC 14882:2011 2.14.5(12)-undefined
ISO/IEC 14882:2011 5.13.5(16)-undefined

参考

MISRA C 2012 7.4

R6.2.7 枚举类型的底层类型不应为 const 或 volatile

ID_uselessQualifier declaration warning

将 enum 或 enum class 的底层类型（underlying type）设为 const 或 volatile 是没有意义的，会被编译器忽略，为语言用法错误。

示例：

```
enum E: const unsigned int  // Non-compliant, ‘const’ is invalid
{
    e0, e1, e2
};
E e = e0;  // ‘e’ is not const
```

应改为：

```
enum E: unsigned int  // Compliant
{
    e0, e1, e2
};
const E e = e0;  // OK, ‘e’ is const
```

依据

ISO/IEC 14882:2011 7.2(2)
ISO/IEC 14882:2011 10.2(2)

R6.2.8 对常量的定义不应为引用

ID_constLiteralReference　　declaration warning

虽然 C++ 语言十分灵活，可以通过多种方式达到同一种目的，但应该选择最简洁且通俗易懂的方式实现。

示例：

```
const int& i = 1024;   // Non-compliant
const int&& j = 1024;  // Non-compliant
```

应改为：

```
const int i = 1024;  // Compliant
const int j = 1024;  // Compliant
```

R6.2.9 禁用 restrict 指针

ID_forbidRestrictPtr　　declaration warning

C 语言中的 restrict 指针要求其他指针不能再指向相同区域，有助于编译器优化，但不符合这种限制时会导致标准未定义的错误，相当于增加了误用的风险，也提高了测试成本。

示例：

```
void foo(int c[]) {
    int *restrict a = &c[0];    // Non-compliant
    int *restrict b = &c[1];    // Non-compliant
    ....
    a = b;                      // Undefined behavior
    ....
}

int bar(int *restrict x, int *restrict y) {  // Non-compliant
    return *x + *y;
}

int baz(int* p) {
    return bar(p, p);  // Undefined behavior
}
```

restrict 指针虽然有助于编译器优化，但应在效率的提高和存在的风险之间进行取舍，非系统库中的代码、改动频繁的代码不建议使用 restrict 指针，而且这种优化大部分情况下也难以真正解决效率的瓶颈问题。

依据

ISO/IEC 9899:1999 6.7.3.1(4 9 11)-undefined
ISO/IEC 9899:2011 6.7.3.1(4 9 11)-undefined

参考

MISRA C 2012 8.14
SEI CERT EXP43-C

R6.2.10 慎用 volatile 关键字

ID_forbidVolatile　　⛔ declaration suggestion

与硬件无关的功能性代码不应使用 volatile 关键字，误用该关键字会引发优化或同步相关的多种问题。

volatile 关键字仅保证编译器不会对其修饰的对象进行优化，确保其有稳定的地址以供读写，在并发编程中容易使人产生误解，其实该关键字在 C/C++ 语言中与同步机制并无关系。

当需要访问某个地址，而其他进程或设备会修改该地址上的数据时，应将其指针声明为 volatile：

```
volatile int* vp = get_hardware_address();
```

如果当前进程不需要主动改变数据，则可加 const 修饰符：

```
const volatile int* cvp = get_hardware_address();
```

这样一来 vp 或 cvp 一直会与该地址对应，利用 *vp 或 *cvp 总会执行从内存的读取操作，不会因为编译器的优化而被忽略，可能需要提供一定的同步措施，但与 volatile 关键字没有关系，这一点与 Java 等语言有较大区别。

另外，利用 volatile 关键字阻止编译优化的特性可实现一些安全措施，参见 ID_unsafeCleanup。

依据

ISO/IEC 14882:2003 7.1.5.1(8)
ISO/IEC 14882:2011 7.1.6.1(7)

参考

C++ Core Guidelines CP.8
C++ Core Guidelines CP.200

6.3 Specifier

R6.3.1 使用 auto 关键字需注意可读性

ID_abusedAuto　　declaration suggestion

---

auto 关键字隐藏了类型名称，在使用时需注意不应降低可读性。

非局部对象不宜用 auto 声明，如接口的返回类型、参数、全局对象等，如果局部对象的类型对程序的行为有显著影响，也不宜用 auto 声明。

示例：

```
auto foo() {
    ....
}

auto bar() {
    auto x = foo();
    ....
    return x;
}

auto obj = bar();  // What the hell is ‘obj’??
```

如果想确定 obj 对象的类型，必须通读所有与之相关的代码，可读性很差。

将代码中所有可以替换成 auto 的标识符全部替换成 auto，其结果是不可想象的，与 Python 等语言不同，C++ 语言存在重载、模板等多种严格依赖于类型的特性，如果类型名称不明确，必然会造成阅读和维护等方面的障碍。

下面给出 auto 关键字的合理用法：

```
Type* a = static_cast<Type*>(ptr);               // Repeated type name
unique_ptr<Type[]> b = make_unique<Type[]>(10);  // Repeated type name
```

重复的类型名称使代码变得繁琐，这种情况使用 auto 是更好的方法：

```
auto* a = static_cast<Type*>(ptr);  // OK
auto b = make_unique<Type[]>(10);   // OK
```

又如：

```
vector<Type> v{ .... };
vector<Type>::iterator i = v.begin();  // Verbose
```

begin 函数返回迭代器是一种常识，且迭代器类型名称往往较长，这种情况应使用 auto：

```
auto i = v.begin();   // OK
```

又如：

```
struct SomeClass {
    struct Sub {
        ....
    };
    Sub foo();
};

SomeClass::Sub SomeClass::foo() {  // Repeated ‘SomeClass’
    ....
}
```

重复的类作用域声明十分繁琐，可用 auto 关键字配合后置返回类型改善：

```
auto SomeClass::foo() -> Sub {  // OK
    ....
}
```

总之，使用 auto 关键字的目的应是提升可读性，而不是单纯地简化代码。

相关

ID_roughAuto

参考

C++ Core Guidelines ES.11

R6.3.2 不应使用已过时的关键字

ID_deprecatedSpecifier　　declaration warning

根据 C++11 标准，register 等关键字已过时，不应再使用，auto 关键字也不可再作为存储类说明符（storage class specifier）。

示例：

```
register int a;  // Non-compliant
auto int b;      // Non-compliant
int foo(register int x);  // Non-compliant
```

本规则对于 C++ 适用，对于 C 可适当放宽要求。

依据

ISO/IEC 14882:2011 D.2(1)-deprecated
ISO/IEC 14882:2011 7.1.6.4

R6.3.3 不应使用多余的 inline 关键字

ID_inlineRedundant　　declaration suggestion

---

constexpr 关键字修饰的函数已经相当于被声明为 inline，不应再重复声明。

示例：

```
inline constexpr int foo(int n) {  // Non-compliant, ‘inline’ is redundant
    return n + 1;
}
```

应改为：

```
constexpr int foo(int n) {  // Compliant
    return n + 1;
}
```

另外，在类声明中实现的函数也相当于被声明为 inline，不应重复声明：

```
class A {
    ....

public:
    inline int foo() {  // Non-compliant, ‘inline’ is redundant
        return 123;
    }

    int bar() {         // Compliant
        return 456;
    }
};
```

依据

ISO/IEC 14882:2011 7.1.5(2)

R6.3.4 extern 关键字不应作用于类成员的声明或定义

ID_invalidExternSpecifier declaration warning

---

extern 关键字作用于类成员的声明或定义是没有意义的，为语言用法错误。

示例：

```
class A {
    void foo();
};

extern void A::foo() {  // Non-compliant, invalid ‘extern’
    ....
}
```

依据

ISO/IEC 14882:2003 9.2(6)
ISO/IEC 14882:2011 9.2(6)
ISO/IEC 14882:2017 12.2(9)

R6.3.5 所有重写的虚函数都应声明为 override 或 final

ID_missingExplicitOverride　　declaration suggestion

将重写的虚函数都声明为 override 或 final，可明显提升代码可读性，并可确保虚函数被有效重写。

示例：

```
class A {
    virtual int foo();
    virtual int bar();
};

class B: public A {
    int foo();          // Non-compliant
    virtual int bar();  // Non-compliant
};
```

如果 B 重写 A 中的 foo 和 bar 两个虚函数，可以按以上代码声明，但如果不看 A 的声明，无法确定 B 中的 foo 和 bar 是对基类虚函数的重写，也看不出 bar 是否是一个新的虚函数。

如果改为：

```
class B: public A {
    int foo() override;  // Compliant
    int bar() override;  // Compliant
};
```

则会清晰很多，而且当重写的函数名、参数或返回值与基类声明不符时，不能通过编译，可及时修正问题。

依据

ISO/IEC 14882:2011 10.3(4 5)

参考

C++ Core Guidelines C.128

R6.3.6 override 和 final 关键字不应同时出现

ID_redundantOverride　　declaration suggestion

---

final 表示不可重写的重写，override 表示可再次重写的重写，这两个关键字不应同时出现。

示例：

```
class D: public B {
public:
   int foo() override final;  // Non-compliant, ‘override’ is redundant
};
```

参考

C++ Core Guidelines C.128

R6.3.7 有 override 或 final 关键字时，不应再出现 virtual 关键字

ID_redundantVirtual　　declaration suggestion

只应在定义新的虚函数时使用 virtual 关键字，重写虚函数应使用 override 或 final 关键字，不应再出现 virtual 关键字。

示例：

```
class A {
public:
    virtual int foo();   // Compliant, a new virtual function
    virtual int bar();   // Compliant, a new virtual function
};

class B: public A {
public:
    virtual int foo() final;     // Non-compliant, ‘virtual’ is redundant
    virtual int bar() override;  // Non-compliant, ‘virtual’ is redundant
};
```

去掉多余的 virtual 关键字使代码更简洁：

```
class B: public A {
public:
    int foo() final;     // Compliant
    int bar() override;  // Compliant
};
```

参考

C++ Core Guidelines C.128

R6.3.8 不应将 union 设为 final

ID_invalidFinal　　declaration warning

标准规定 union 不可作为基类，所以将 union 声明为 final 是没有意义的，属于语言运用错误。

示例：

```
union U final  // Non-compliant, meaningless
{
    ....
};
```

依据

ISO/IEC 9899:2011 9.5(2)

R6.3.9 inline、virtual、static、typedef 等关键字应出现在类型名的左侧

ID_badSpecifierPosition　　declaration suggestion

语言允许 inline、virtual、static、typedef 等关键字可以出现在类型名的左侧，也可以出现在其右侧，甚至可以出现在基本类型名的中间，为了提高可读性，应对其位置进行统一规范。

示例：

```
struct A {
    long long typedef ll_t;     // Non-compliant
    bool static foo();          // Non-compliant
    char friend bar();          // Non-compliant
    unsigned inline int baz();  // Non-compliant
};
```

应统一规定出现在类型名的左侧：

```
struct A {
    typedef long long ll_t;     // Compliant
    static bool foo();          // Compliant
    friend char bar();          // Compliant
    inline unsigned int baz();  // Compliant
};
```

本规则对下列关键字有同样的要求：

```
inline、virtual、explicit、
register、static、thread_local、extern、mutable、
friend、typedef、constexpr
```

对于 const 和 volatile 也建议出现在类型名的左侧。

相关

ID_sandwichedModifier
ID_badQualifierPosition

6.4 Declarator

R6.4.1 用 auto 声明指针或引用时应显式标明 *、& 等符号

ID_roughAuto　　declaration suggestion

用 auto 声明指针时显式标明 * 号有利于提高代码可读性，否则会使人误以为是某种非指针的对象。在声明引用时必须显式标明 & 或 && 号，否则成为对象声明，导致逻辑错误或造成不必要的复制开销。

示例：

```
int* foo();
int& bar();

auto p = foo();   // Bad
auto* q = foo();  // Good

auto r = bar();   // Becareful, 'r' is not a reference

for (auto e: container) {  // Is it necessary to copy elements?
    ....
}
```

例中 p 为指针，但看起来像是个对象，bar 返回引用，但 r 并不是引用，在遍历容器时，e 是容器元素的复本，这些问题可能会造成错误，需谨慎对待。

R6.4.2 禁用可变参数列表

ID_forbidVariadicFunction ⛔ declaration warning

可变参数列表对参数的类型和数量缺乏有效的限定和控制，是公认的不安全因素。

示例：

```
string format(const char* fmt, ...);  // Non-compliant
```

假设 format 函数与 sprintf 函数功能相似，由参数 fmt 设定格式，将其他参数转为字符串后依次替换 fmt 中的占位符并返回结果。设 '@' 和 '$' 为占位符，分别对应字符串和整数，如调用 format("@: $", "value", 123) 则返回字符串 "value: 123"。

如果用可变参数列表实现：

```
string format(const char* fmt, ...) {
    string res;
    va_list vl;
    va_start(vl, fmt);
    for (auto* p = fmt; *p; p++) {
        stringstream ss;
        switch(*p) {
            case '@': ss << va_arg(vl, char*); break;
            case '$': ss << va_arg(vl, int); break;
            default:  ss << *p; break;
        }
        res.append(ss.str());
    }
    va_end(vl);
    return res;
}
```

例中 va_start、va_arg、va_end 是可变参数列表的标准支持，这种方法只能在运行时以 fmt 为依据获取后续参数，当实际参数与 fmt 不符时会造成严重问题，单纯地要求代码编写者小心谨慎是不可靠的，改用更安全的方法才是明智的选择。

在 C++ 代码中可采用“模板参数包”来实现这种功能：

```
template <class T, class ...Args>
void get_argstrs(vector<string>& vs, const T& arg, const Args& ...rest) {
    ostringstream oss;
    oss << arg;
    vs.emplace_back(oss.str());
    if constexpr(sizeof...(rest) > 0) {
        get_argstrs(vs, rest...);
    }
}

template <class ...Args>
string format(const char* fmt, const Args& ...args) {  // Compliant
    string res;
    if constexpr(sizeof...(args) > 0) {
        vector<string> vs;
        const size_t n = strlen(fmt);
```

```
        get_argstrs(vs, args...);
        for (size_t i = 0, j = 0; i < n; i++) {
            if ((fmt[i] == '@' || fmt[j] == '$') && j < vs.size()) {
                res.append(vs[j++]);
            } else {
                res.push_back(fmt[i]);
            }
        }
    }
    return res;
}
```

示例代码用 get_argstrs 函数递归地将参数都转为 string 对象存入容器，再将 fmt 中的 '@' 和 '$' 依次替换成容器中的字符串，实际上这种实现是可以不区分 '@' 和 '$' 的，这个过程中参数的个数和类型是可以由代码主动判断的，如果参数不能转为字符串则不会通过编译，如果参数个数与占位符不符也容易作出处理。

“模板参数包”、“constexpr”等特性是 C++ 语言在编译理论上的重大突破，合理运用这些特性可以有效提升代码的安全性和可维护性。

参考

C++ Core Guidelines ES.34
C++ Core Guidelines F.55
MISRA C 2004 16.1
MISRA C++ 2008 8-4-1

R6.4.3 禁用柔性数组

ID_forbidFlexibleArray　　⊖ declaration suggestion

柔性数组（flexible array）一般是指结构体最后不完整定义的数组成员，表示不占用空间的指针，这种数组在 C99 中有所定义，但不在 C++ 标准之中，在 C++ 代码中不应使用。

示例：

```
struct A {
    int len;
    int dat[];  // Non-compliant
};

A* cpy(const A* p) {
    A* a = (A*)malloc(sizeof(A) + p->len * sizeof(int));
    *a = *p;   // Error, only p->len is copied
    return a;
}
```

例中 *a=*p 这种拷贝赋值运算会漏掉数组的内容，而且数组不会计入 sizeof 的结果，易引起意料之外的错误，所以在 C 语言中也不建议使用这种柔性数组。

依据

ISO/IEC 9899:1999 6.7.2.1(16)

参考

MISRA C 2012 18.7

R6.4.4 接口的参数或返回值不应被声明为 void*

ID_forbidFunctionVoidPtr　　⛔ declaration warning

与接口相关的数据类型应保持精确，不应将参数或返回值声明为 void*。

在 C++ 语言中，如果参数或返回值需要面对多种不同类型的数据，应合理使用重载或模板机制。

示例：

```
class A {
public:
    void* foo();      // Non-compliant
    void bar(void*);  // Non-compliant
};
```

例中 foo 和 bar 函数的返回值以及参数是不符合要求的。

C 语言中存在大量的库函数不符合本规则要求，在 C++ 语言中应避免使用，如：

```
int buf[123];
memset(buf, 0, 123);  // Logic error, should be '123 * sizeof(int)'
```

例中 memset 函数的第一个形式参数就是 void* 型，只能通过更底层的二进制方式访问对象序列，是一种对类型设计的破坏，应改用 STL 标准库提供的方法：

```
int buf[123];
std::fill_n(buf, 123, 0);  // Safe and brief
```

改用类型明确的方法可以使很多问题在编译期得到控制。

例外：
C++ 语言规定 new 运算符的返回类型为 void*，delete 运算符的参数类型为 void*，这些情况可被排除。

相关

ID_forbidMemberVoidPtr

参考

C++ Core Guidelines I.4

R6.4.5 类成员不应被声明为 void*

ID_forbidMemberVoidPtr ⛔ declaration warning

与接口相关的数据类型应保持精确，不应将类成员声明为 void*，尤其是非 private 成员，更不应声明为 void*。

在 C++ 语言中，如果成员需要面对多种不同类型的数据，应合理使用模板机制。

示例：

```
class A {
public:
    void* dat;  // Non-compliant
    ....
};
```

应改为：

```
template <class T>
class A {
public:
    T* method_about_dat();
private:
    T* dat;     // Compliant
    ....
};
```

相关

ID_forbidFunctionVoidPtr

参考

C++ Core Guidelines I.4

R6.4.6 局部数组的长度不应过大

ID_unsuitableArraySize　　declaration warning

局部数组的长度过大增加函数堆栈的压力，易导致溢出错误。

示例：

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

局部数组在栈上分配空间，无法控制失败情况，大型数组应在堆上分配，或优化算法降低空间成本：

```
void foo() {
    int* arr = (int*)malloc(1024 * 1024 * 1024 * sizeof(int));  // Compliant
    if (arr) {
        ....     // Your business
    } else {
        ....     // Handle allocation failures
    }
}
```

量化评估程序需要的堆栈空间是产品设计的重要环节，相关的评审与测试也需要落实。

配置

maxLocalArraySize：局部数组的长度上限，超过则报出

参考

CWE-770
SEI CERT MEM05-C

R6.4.7 不建议将类型定义和对象声明写在一个语句中

ID_mixedTypeObjDefinition　　declaration suggestion

将类型定义和对象声明写在一个语句中可读性较差。

示例：

```
struct T {
    ....
} obj, *ptr;  // Bad
```

参考

C++ Core Guidelines C.7

R6.4.8 不应将函数或函数指针和其他声明写在同一个语句中

ID_mixedDeclarations　　declaration suggestion

每条语句只应声明一个函数或函数指针，否则可读性较差。

示例：

```
int* foo(int), bar(0), (*baz)(char);  // Non-compliant, very bad
```

例中 foo 是函数，bar 是整数，baz 是函数指针，这种混在一起的声明是非常混乱的。

应分开声明：

```
int* foo(int);    // Compliant
int bar = 0;      // Compliant
int (*baz)(char); // Compliant
```

相关

ID_tooManyDeclarators

参考

C++ Core Guidelines ES.10

R6.4.9 在一个语句中不应声明过多对象或函数

ID_tooManyDeclarators　　declaration suggestion

在一个语句中不应声明过多对象或函数，建议在每个语句中只声明一个对象或函数，提高可读性也可减少笔误。

示例：

```
int* a, b[8], c, d(int), e = 0;  // Bad
```

例中只有 a 是指针，也只有 e 被初始化，d 为函数，应分开声明。

配置

maxDeclaratorCount：一个声明语句能包含的对象个数上限，超过则报出

参考

C++ Core Guidelines ES.10
MISRA C++ 2008 8-0-1

6.5 Object

R6.5.1 不应产生无效的临时对象

ID_inaccessibleTmpObject　　declaration error

无名且不受控制的临时对象在构造之后会立即析构，在逻辑上没有意义，往往意味着错误。

示例：

```
class A {
    int a;

public:
    A() {
        A(0);  // Non-compliant, just created an inaccessible temporary object
    }

    A(int x): a(x) {
    }
};
```

例中 A(0); 只生成了一个无效的临时对象，成员并没有被正确初始化，应改为 this->A::A(0); 等形式。

又如：

```
class LockGuard { .... };

void fun() {
    LockGuard();   // Non-compliant, meaningless
    ....
}
```

设 LockGuard 是某种锁，LockGuard(); 只生成了一个临时对象，该对象会立即析构，起不到作用，这也是一种常见的错误。

应改为：

```
void fun() {
    LockGuard guard;  // Compliant
    ....
}
```

参考

CWE-665
C++ Core Guidelines ES.84

R6.5.2 不应出现不会被用到的局部声明

ID_invalidLocalDeclaration　　declaration warning

不会被用到的局部声明是没有意义的，往往意味着笔误或者代码功能不完整。

示例：

```
int foo(int a, int b) {
    if (a)
        int b = a + 1;  // Non-compliant
    return b;
}
```

例中 if 作用域中的最后一个元素是对象的声明，是没有意义的，这种情况也是较为常见的笔误。

应改为：

```
int foo(int a, int b) {
    if (a) {
        b = a + 1;  // OK
    }
    return b;
}
```

R6.5.3 对象初始化不可依赖自身的值

ID_selfDependentInitialization　　declaration error

对象初始化依赖自身的值属于逻辑错误，也是常见的笔误。

示例：

```
void foo(int i) {
    if (i > 0) {
        int i = i + 1;  // Non-compliant
        ....
    }
}
```

例中局部变量 i 的初始化依赖自身的值，这种问题往往是错误地定义了与外层作用域中名称相同的对象。

应改为：

```
void foo(int i) {
    if (i > 0) {
        int j = i + 1;  // OK
        ....
    }
}
```

R6.5.4 参与数值运算的 char 变量需显式声明 signed 或 unsigned

ID_plainNumericChar　　declaration warning

没有 signed 或 unsigned 限制的 char 类型，是否有符号由具体的编译器决定。

示例：

```
bool foo(char c) {          // Non-compliant
    return c < 180;         // May be always true
}

void bar() {
    char c = 180;           // Non-compliant
    printf("%d", 2 * c);    // What is output?
}
```

这段代码可移植性较差，foo 在某些环境中可能只会返回 true，而 bar 可能输出 -152，也可能输出 360。

char 类型在 PC 桌面、服务端等环境中一般是有符号的，在移动端或嵌入式系统中往往是无符号的，需明确其具体实现。

应改为：

```
bool foo(unsigned char c) { // Compliant
    return c < 180;
}

void bar() {
    unsigned char c = 180;  // Compliant
    printf("%d", 2 * c);    // 360
}
```

依据

ISO/IEC 14882:2003 3.9.1(1)-implementation
ISO/IEC 14882:2011 3.9.1(1)-implementation

参考

MISRA C++ 2008 5-0-11
SEI CERT INT07-C

R6.5.5 字节的类型应为 unsigned char

ID_plainBinaryChar declaration warning

字节等二进制概念不应受符号位干扰，应声明为 unsigned char。

示例：

```
typedef char byte;  // Non-compliant
byte buf[100];
FILE* fp = fopen("foo", "rb");
fread(buf, 1, 100, fp);
if (buf[0] == 0xff) {  // May be always false
    ....
}
if (buf[1] << 1) {     // May cause undefined behavior
    ....
}
```

char 类型的符号由实现定义，有符号的 char 变量在数值计算、位运算等方面很容易产生意料之外的结果。

应改为：

```
typedef unsigned char byte;  // Compliant
```

这样做也可有效区分二进制数据与字符串，提高可读性。

相关

ID_plainNumericChar
ID_bitwiseOperOnSigned

依据

ISO/IEC 14882:2003 3.9.1(1)-implementation
ISO/IEC 14882:2011 3.9.1(1)-implementation

6.6 Parameter

R6.6.1 函数原型声明中的参数应具有合理的名称

ID_missingParamName　　declaration suggestion

参数的名称是其用途的直接说明，合理的名称可显著提高可读性。

示例：

```
char* strstr(const char* haystack, const char* needle);  // Good
```

这是标准库函数 strstr 的原型声明，利用形象的比喻，表示在 haystack 中查找 needle。

如果将声明改为如下形式，就令人费解了：

```
char* strstr(const char*, const char*);  // Bad
char* strstr(const char* a, const char* b);  // Bad
```

参考

MISRA C 2004 16.3
MISRA C 2012 8.2

R6.6.2 不建议虚函数的参数有默认值

ID_deprecatedDefaultArgument　　declaration suggestion

虚函数参数的默认值不受多态规则控制，通过基类指针或引用调用派生类重写的虚函数时，默认值仍采用基类中的定义，易造成混淆，建议虚函数参数不使用默认值。

示例：

```
class A {
public:
    virtual int foo(int i = 0);  // Bad
};
```

应尽量去掉默认参数值，或改用重载函数的方式：

```
class A {
public:
    virtual int foo();  // OK
    virtual int foo(int i);  // OK
};
```

相关

ID_inconsistentDefaultArgument

依据

ISO/IEC 14882:2003 8.3.6(10)
ISO/IEC 14882:2011 8.3.6(10)
ISO/IEC 14882:2017 11.3.6(10)

参考

CWE-628
C++ Core Guidelines C.140
MISRA C++ 2008 8-3-1

R6.6.3 虚函数参数的默认值应与基类中声明的一致

ID_inconsistentDefaultArgument　　declaration error

虚函数参数的默认值不受多态规则控制，通过基类指针或引用调用派生类重写的虚函数时，默认值仍采用基类中的定义。

示例：

```
class A {
public:
    virtual int foo(int i = 0) {
        return i;
    }
};

class B: public A {
public:
    int foo(int i = 1) override {  // Non-compliant
        return i + 1;
    }
};

A* p = new B;
cout << p->foo() << '\n';  // What is output?
```

输出 1，这种虚函数的非多态行为是非常令人困惑的。

相关

ID_deprecatedDefaultArgument

依据

ISO/IEC 14882:2003 8.3.6(10)
ISO/IEC 14882:2011 8.3.6(10)
ISO/IEC 14882:2017 11.3.6(10)

参考

CWE-628
C++ Core Guidelines C.140
MISRA C++ 2008 8-3-1

R6.6.4 不应将数组作为函数的形式参数

ID_invalidParamArraySize　　declaration warning

在形式参数中对数组大小的声明起不到实际的限制作用。

示例：

```
int foo(int a[100]);  // Non-compliant

int bar() {
    int a[50] = {};
    return foo(a);    // It can be compiled
}
```

在 C++ 语言中可改为数组的引用：

```
void foo(int (&a)[100]);   // Compliant

template <size_t size>
void foo(int(&a)[size]) {  // Compliant
    ....
}
```

依据

ISO/IEC 9899:1999 6.7.5.3(7)
ISO/IEC 9899:2011 6.7.6.3(7)

参考

C++ Core Guidelines I.13
C++ Core Guidelines R.14
MISRA C++ 2008 5-2-12

R6.6.5 C 代码中参数列表如果为空应声明为“(void)”

ID_missingVoid　　declaration warning

在 C 语言中，如果函数的参数列表声明为空括号，表示函数的参数还没有声明，而不是表示没有参数，这很容易使人误解，所以在 C 代码中没有参数的参数列表应声明为“(void)”。

示例：

```
// In a.h
int foo();  // Non-compliant

// In a.c
#include "a.h"

int foo(int a) {
    return a + 1;
}

// In main.c
```

```
#include <stdio.h>
#include "a.h"

int main() {
    printf("%d\n", foo(1));  // Output: 2
    printf("%d\n", foo());   // Can be compiled, but what is output?
}
```

例中 foo(1) 和 foo() 两种调用都可以通过编译，然而声明与实现不一致的问题总是令人困惑的，如果明确将参数声明为 void 或 int a 则可以解决这种问题。

应改为：

```
int foo(void);   // Compliant, 'foo(1)' cannot be compiled
```

或者：

```
int foo(int a);  // Compliant, 'foo()' cannot be compiled
```

相关

ID_superfluousVoid

依据

ISO/IEC 9899:2011 6.7.6.3(14)
ISO/IEC 9899:2011 6.11.6(1)

参考

MISRA C 2004 16.5

R6.6.6 C++ 代码中参数列表如果为空不应声明为“(void)”

ID_superfluousVoid　　declaration suggestion

与 C 语言不同，在 C++ 中空括号和“(void)”均表示没有参数，所以应采用更简洁的方式。

示例：

```
struct A {
    int foo(void);  // Verbose
    int bar();      // OK
};
```

相关

ID_missingVoid

依据

ISO/IEC 14882:2003 C.1.6 Clause 8
ISO/IEC 14882:2011 C.1.7 Clause 8
ISO/IEC 14882:2017 C.1.7 Clause 11

参考

C++ Core Guidelines NL.25

R6.6.7 声明数组参数的大小时禁用 static 关键字

ID_forbidStaticArrSize ⊖ declaration warning

C 语言规定数组作为形式参数时，可用 static 关键字修饰大小，要求传入数组的大小不能小于由 static 关键字修饰的值，有助于编译器优化，但不符合这种限制时会导致标准未定义的错误，相当于增加了误用的风险，也提高了测试成本。

示例：

```
int foo(int a[static 5], int n) {  // Non-compliant
    int i;
    int s = 0;
    for (i = 0; i < n; i++) {
        s += a[i];
    }
    return s;
}

int bar() {
    int a[3] = {1, 2, 3};
    return foo(a, 3);      // Undefined behavior
}
```

这种机制虽然有助于编译器优化，但应在效率的提高和存在的风险之间进行取舍，非系统库中的代码、改动频繁的代码不建议使用这种机制，而且这种优化大部分情况下也难以真正解决效率的瓶颈问题。

依据

ISO/IEC 9899:1999 6.7.6.3(7)
ISO/IEC 9899:2011 6.7.6.3(7)

参考

MISRA C 2012 17.6

6.7 Function

R6.7.1 派生类不应重新定义与基类相同的非虚函数

ID_nonVirtualOverride declaration warning

派生类不应重新定义与基类相同的非虚函数，否则与多态机制相矛盾，易造成意料之外的问题。

示例：

```
class A {
public:
    int foo() const { return 0; }
};

class B: public A {
public:
    int foo() const { return 1; }  // Non-compliant
};

int bar(A* a) {
    return a->foo();
}
```

如果将 B 类型的指针传入 bar 函数，将执行 A::foo，然而参数实际指向的是 B 类型的对象，但 B::foo 不会被执行，这就形成了逻辑上的矛盾，造成意料之外的问题。

参考

Effective C++ item 36

R6.7.2 拷贝赋值、移动赋值运算符应返回所属类的非 const 引用

ID_nonStdAssignmentRetType declaration warning

拷贝赋值、移动赋值运算符应返回所属类的非 const 引用，便于调用者使用并满足泛型编程的要求。

对赋值运算符的合理重载，可以使对象的赋值表达式作为子表达式灵活地出现在各种语句中，这也是“泛型程序设计”的必要条件，使算法的代码实现既可以适应普通变量，也可以适应自定义对象。如果类的对象与标准模板库相关，其赋值运算符应满足本规则的要求，否则无法满足连续赋值等语法要求，在标准模板库的使用上会受到限制。

本规则对 +=、-= 等复合赋值运算符也有相同的要求。

示例：

```
struct A {
    A& operator = (const A&);    // Compliant
    A& operator = (A&&);         // Compliant
};

struct B {
    void operator = (const B&);  // Non-compliant, should return ‘B&’
    void operator = (B&&);       // Non-compliant, should return ‘B&’
};
```

参考

C++ Core Guidelines F.47
C++ Core Guidelines C.60
C++ Core Guidelines C.63

R6.7.3 拷贝赋值运算符的参数应为同类对象的 const 左值引用

ID_nonStdCopyAssignmentParam　　declaration warning

拷贝赋值运算符的参数不应按值传递，否则会造成不必要的复制，以及“对象切片”等问题。

示例：

```
struct A {
    A& operator = (A);  // Non-compliant
};
```

应改为：

```
struct A {
    A& operator = (const A&);  // Compliant
};
```

参考

C++ Core Guidelines C.60

R6.7.4 移动赋值运算符的参数应为同类对象的非 const 右值引用

ID_nonStdMoveAssignmentParam　　declaration warning

移动赋值运算符的参数不可为 const 右值引用，否则将失去移动赋值的意义。

示例：

```
class A {
    char* p;

public:
    A& operator = (const A&& a) {  // Non-compliant
        free(p);
        p = copy(a.p);             // Not necessary
        return *this;
    }

    ....
};
```

例中赋值运算符先释放持有的资源，再复制 a 的资源，不是真正的移动赋值，仍是一种低效实现。应将 a.p 与 this->p 交换，省去复制过程，并使原有资源由 a 的析构函数释放，才是真正意义上的移动赋值：

```
class A {
    char* p;

public:
    A& operator = (A&& a) noexcept {  // Compliant
        char* tmp = p;
        p = a.p;
        a.p = tmp;
        return *this;
    }

    ....
};
```

参考

C++ Core Guidelines C.63

R6.7.5 不应重载取地址运算符

ID_overloadAddressOperator　　declaration suggestion

取地址运算符（一元 & 运算符），重载之后可以返回任意地址，极易误用。

获取不完整类型的对象地址时，如果其完整类型重载了取地址运算符，会导致标准未定义的行为。

示例：

```
struct X;  // Incomplete type

X* foo(X& x) {
    return &x;  // Undefined behaviour
}

struct X {
    X* operator &() {  // Non-compliant
        return nullptr;
    }
};

X* bar(X& x) {
    return &x;  // Call 'X::operator&'
}
```

例中 foo 函数存在标准未定义的问题，可能会返回 x 对象的实际地址，而 bar 函数会调用重载了的取地址运算符，这是一种混乱的局面。

依据

ISO/IEC 14882:2003 5.3.1(4)-undefined
ISO/IEC 14882:2011 5.3.1(5)-undefined

参考

C++ Core Guidelines C.166
MISRA C++ 2008 5-3-3

R6.7.6 不应重载逗号运算符

ID_overloadComma　　declaration suggestion

逗号表达式意在从左至右依次执行各子表达式，但重载逗号运算符会打破这一规则，易造成意料之外的结果。

示例：

```
class A { .... };

A& foo(int);
A& operator , (int, A&);  // Non-compliant

int bar(int i) {
    ++i, foo(i);   // Disordered
}
```

例中逗号运算符被重载后变成了一个函数，++i 和 foo(i) 变成了函数参数，函数参数的求值顺序在标准中是未声明的，foo(i) 很有可能会先被求值，++i 则失去了意义。

另外，也不应重载逻辑运算符，参见 ID_overloadLogicOperator。

相关

ID_overloadLogicOperator

依据

ISO/IEC 14882:2011 5.2.2(8)

参考

MISRA C++ 2008 5-2-11

R6.7.7 不应重载“逻辑与”和“逻辑或”运算符

ID_overloadLogicOperator　　declaration suggestion

对“逻辑与”、“逻辑或”等运算符的重载会影响效率，甚至造成不符合预期的结果。

C/C++ 标准明确规定了内置逗号、逻辑与、逻辑或等运算符的子表达式求值顺序。对于逻辑表达式，从左到右计算子表达式的值，当可以确定整个表达式的值时立即结束计算，如果还有其他子表达式未求值也不再计算了，这种规则称为“短路规则”，意在提高效率，然而运算符的重载却打破了这一规则。

示例：

```
class A {
    int i;

public:
    A(int x = 0): i(x) {
    }

    bool valid() const {
        return i != 0;
    }
```

```
    A& assign(const A& a) {
        i = a.i;
        return *this;
    }
};

bool operator && (const A& a, const A& b) {  // Non-compliant
    return a.valid() && b.valid();
}
```

注意表达式（设 a 和 b 为 A 的对象）：

```
b && a.assign(b)
```

按常理，此表达式的意思应该是如果 b 在某种意义上“有效”，就将 b 赋给 a，所以 b 的值应先被求出，但由于 && 被重载成了一个函数，其左右子表达式成了函数的参数，“短路规则”不再有效，而且参数的求值顺序在标准中是未声明的，所以常规逻辑子表达式的求值顺序无法得到保证。目前 MSVC、g++ 等主流编译器默认都是从右到左计算参数的值，例中 a.assign(b) 会先被执行，造成完全不符合预期的结果。

解决方法：

```
class A {
    ....

public:
    operator bool() const {
        return valid();
    }
};
```

去掉对 && 的重载，在 A 中定义 bool 类型转换运算符，既可保证“短路规则”，又可保证求值顺序。

相关

ID_overloadComma

依据

ISO/IEC 14882:2011 5.2.2(8)

参考

MISRA C++ 2008 5-2-11

R6.7.8 拷贝赋值、移动赋值运算符不应为虚函数

ID_virtualAssignment declaration warning

拷贝或移动赋值运算符的返回类型应为所属类的非 const 引用，这类运算符即使是虚函数也不便于被重写。

示例：

```
class A {
public:
    virtual A& operator = (const A&);  // Non-compliant
};

class B: public A {
public:
    virtual B& operator = (const B&);  // Not override
    virtual A& operator = (const A&);  // Override, but too complex
};
```

相关

ID_nonStdAssignmentRetType

参考

C++ Core Guidelines C.60
C++ Core Guidelines C.63

R6.7.9 比较运算符不应为虚函数

ID_virtualComparison declaration warning

重载的比较运算符很难正确触发 C++ 的多态机制，将其设为虚函数很可能引发意料之外的错误。

运算符 ==、!=、<、>、 <=、>= 均受本规则限制。

示例（选自 C++ Core Guidelines）：

```
class B {
  string name;
  int number;
  virtual bool operator==(const B& a) const
  {
    return name == a.name && number == a.number;
  }
  // ...
};
```

```
class D : B {
  char character;
  virtual bool operator==(const D& a) const
  {
    return name == a.name && number == a.number && character == a.character;
  }
  // ...
};

B b = ...
D d = ...
b == d;    // compares name and number, ignores d's character
d == b;    // error: no == defined
D d2;
d == d2;   // compares name, number, and character
B& b2 = d2;
b2 == d;   // compares name and number, ignores d2's and d's character
```

参考

C++ Core Guidelines C.87

R6.7.10 final 类中不应声明虚函数

ID_virtualInFinal　　declaration warning

---

final 类不再产生派生类，其中的虚函数也不会再被重写，故不应声明虚函数。

示例：

```
class A final {
public:
    virtual int foo();  // Non-compliant, a new virutal function in final class
};

class B {
public:
    virtual int bar();
};

class C final: public B {
public:
    virtual int bar();  // Non-compliant, use keyword ‘override’ or ‘final’
};
```

依据

ISO/IEC 9899:2011 9(3)

6.8 Bitfield

R6.8.1 位域长度不应超过类型约定的大小

ID_exceededBitfield declaration warning

位域长度不应超过类型约定的大小，否则没有意义且会造成不必要的空间浪费。

C 标准规定位域长度不可超过类型约定的大小，但 C++ 标准规定可以超过，超过的部分作为“padding bits”不参与数据的存储。

示例：

```
struct A {
    int x: 50;  // Non-compliant
};

int main() {
    A a;
    cout << sizeof(a) << '\n';  // What is output?
}
```

输出 8，例中成员 x 自身的位域长度仍为 32，而多出来的大小会形成一个不可访问的填充值。

依据

ISO/IEC 9899:2011 6.7.2.1(4)
ISO/IEC 14882:2011 9.6(1)

R6.8.2 有符号变量的位域长度不应为 1

ID_singleSignedBitfield declaration warning

有符号变量的位域长度如果为 1 表示只有一个比特位，而该比特位是符号位，极易造成意料之外的错误。

本规则不针对匿名成员。

示例：

```
struct X {
    int a: 1;           // Non-compliant
    unsigned int b: 1;  // Compliant
    int c: 2;           // Compliant
};

int main() {
    X x;
    x.a = 1; cout << x.a << '\n';  // What is output?
    x.b = 1; cout << x.b << '\n';
    x.c = 1; cout << x.c << '\n';
}
```

按常规思维，x.b 和 x.c 为 1 与预期相符，x.a 预期是 1，但实际是 -1。

参考

MISRA C++ 2008 9-6-4

R6.8.3 不应对枚举变量声明位域

ID_forbidEnumBitfield　　⛔ declaration warning

枚举变量的类型可以是有符号整数，符号位与位域结合易导致意料之外的错误，且不利于枚举类型的扩展。

示例：

```
enum E {
    A, B, C, D
};

struct X {
    E e: 2;  // Non-compliant
};

int main() {
    X x;
    x.e = D;
    if (x.e == D) {   // What is output?
        cout << "OK";
    } else {
        cout << "Oops";
    }
}
```

输出 Oops，例中 E 各枚举项的取值范围是 [0,3]，按常规思维，位域长度为 2 即可满足这个范围，然而符号位的存在否定了这一点，导致 x.e 用 D 赋值后再与 D 比较竟然得到不相等的结果（因为 D 的值为 3 而 x.e 的值为 -1）。

相关

ID_singleSignedBitfield

参考

MISRA C++ 2008 9-6-3

R6.8.4 禁用位域

ID_forbidBitfield　　⛔ declaration suggestion

引入位域的本意是为了节省空间，然而位域改变了变量约定俗成的取值范围，易造成理解上的偏差，也会造成维护困难。

位域与“引用”等 C++ 概念有冲突，而且 C++ 标准在位域的内存分配和数据对齐等方面定义的不够充分，存在很多由实现定义的内容，所以应改用某种更有效的算法达到节省空间或时间的目的。

示例：

```
struct A {
    int a: 3;   // Non-compliant
    int b;      // Compliant
};
```

相关

ID_exceededBitfield
ID_singleSignedBitfield
ID_forbidEnumBitfield

依据

ISO/IEC 14882:2003 9.6(1)-implementation
ISO/IEC 14882:2003 9.6(3)
ISO/IEC 14882:2011 9.6(1)-implementation
ISO/IEC 14882:2011 9.6(3)
ISO/IEC 14882:2017 12.2.4(1)-implementation
ISO/IEC 14882:2017 12.2.4(3)

6.9 Complexity

R6.9.1 不建议采用复杂的声明

ID_complexDeclaration　　declaration suggestion

复杂的声明可读性较差，容易造成理解上的偏差。

对于：

- 函数指针的数组
- 返回函数指针、数组指针的函数
- 以函数指针、数组指针为参数的函数

应先将各子类型取别名，再用简单声明的方式书写。

示例：

```
int (*foo(int))(bool);  // Bad, returns a function pointer
int (*foo(char))[123];  // Bad, returns an array pointer
```

例中声明的是两个函数，但看起来像是函数指针，而且参数列表也显得混乱。

应改为：

```
typedef int(*funptr)(bool);
typedef int(*arrptr)[123];

funptr foo(int);   // Good
arrptr foo(char);  // Good
```

另外，指针的星号个数不宜超过两个，否则意味着指针的解引用逻辑过于复杂，如：

```
T *** x;  // Bad
T * const * * const * y;  // Horrible
```

其中 T 为任意类型，如果发现这种指针，意味着需要改进对相关数据的访问方式。

6.10 Other

R6.10.1 不应存在没有用到的标签

ID_labelNotUsed　　declaration warning

没有用到的标签意味着 goto 语句的缺失，也可能是残留代码，应当去除。

示例：

```
int foo() {
    int i = 0;
lab:                // Non-compliant
    return bar(i);
}
```

标签只应与 goto 语句对应，不可有其他用途，如：

```
int bar(int x) {
    int i = 0;
strange_comment:    // Non-compliant
    return x + i;
}
```

例中标签被当作注释，这是一种怪异的用法，也可能会干扰编译器的优化。

参考

MISRA C 2012 2.6

R6.10.2 不应存在未被使用的本地 static 函数

ID_staticNotUsed　　declaration warning

未被使用的本地 static 函数得不到任何执行机会，应删除或修正调用关系。

示例：

```
static int foo();   // Compliant
static int bar();   // Non-compliant, unused

int main() {
    return foo();
}
```

相关

ID_unreachableCode

参考

MISRA C++ 2008 0-1-10

R6.10.3 不应存在未被使用的 private 成员

ID_privateNotUsed declaration warning

未被使用的 private 成员没有意义，应删除或修正引用关系。

示例：

```
struct A {
    int foo() { return 1; }     // Compliant, public or protected

private:
    int bar;                    // Non-compliant, unused
    int foo(int) { return 0; }  // Non-compliant, unused
};
```

相关

ID_unreachableCode

参考

MISRA C++ 2008 0-1-10

R6.10.4 避免使用 std::auto_ptr

ID_deprecatedAutoPtr declaration warning

---

std::auto_ptr 在 C++11 标准中已被废弃，应使用 std::unique_ptr。

std::auto_ptr 在转移资源所有权等方面易被误用，std::unique_ptr 在相关方面有更严格的限制。

示例：

```
class T { .... };
void bar(auto_ptr<T> p);

auto_ptr<T> a(new T);
auto_ptr<T> b;
....
b = a;      // ‘a’ is invalid after the assignment
bar(b);     // ‘b’ is invalid after this call
....        // Undefined behavior if dereference ‘a’ or ‘b’
```

auto_ptr 对象的赋值或按值传参都会引起资源所有权的转移，如 b = a 表示 a 的资源被 b 占有，foo(b) 表示 b 的资源被参数占有，之后再对 a 或 b 解引用就会造成错误，这种方式很容易被人误解，C++11 标准已弃用。

unique_ptr 禁止资源所有权的隐式转移，语义更为明确：

```
unique_ptr<T> a = make_unique<T>();
unique_ptr<T> b;
....
b = move(a);   // OK, explicit moving
foo(b);        // Complie error
```

unique_ptr 对象必须通过 move 显式转移资源所有权，否则无法通过编译。

依据

ISO/IEC 14882:2011 D.10-deprecated

参考

C++ Core Guidelines R.20

7. Exception

R7.1 确保异常的安全性

ID_exceptionUnsafe　　exception warning

当产生异常时，保证：

- 相关资源不会泄漏
- 相关对象处于正确状态

是 C++ 异常机制可以正确工作的重要基础。

示例：

```
void foo() {
    lock();
    procedure_may_throw();  // Unsafe
    unlock();
}
```

设 lock 是某种获取资源的操作，unlock 是释放资源的操作，procedure_may_throw 是可能抛出异常的过程，那么 foo 函数就不是异常安全的，一旦有异常抛出会导致死锁或泄露等问题。

应保证资源从分配到回收的过程不被异常中断，采用对象化管理方法，使分配和回收得以自动完成：

```
void foo() {
    LockOwner object;
    procedure_may_throw();  // Safe
}
```

将 lock 和 unlock 分别由 object 的构造和析构函数完成，即使 procedure_may_throw 抛出异常，相关资源也可被自动回收，实现了异常安全，资源的对象化管理方法可参见 ID_ownerlessResource。

异常安全的另一个重要方面是抛出异常时应保证相关对象的状态是正确的，事务或算法在处理对象时可能要分多个步骤处理对象的多个成员，要注意中途抛出异常会造成数据不一致等问题。

```
class X {
    T a, b;

public:
    void foo() {
        proc(a);
        // ... If throw an exception ...
        proc(b);
    }
};
```

设 a 和 b 是两个密切相关的成员，如账号和金额等，foo 是一个处理事务的函数，如果在中途抛出异常就会使对象处于错误的状态，解决方法可以考虑"复制 - 交换"模式，如：

```
class X {
    T a, b;

public:
    void foo() {
        X copy(*this);
        proc(copy.a);
        proc(copy.b);
        this->swap(copy);
    }

    void swap(X& v) noexcept {
        ....
    }
};
```

事务先处理对象的副本，处理成功后交换副本与对象的数据，交换过程需要保证不抛出异常，这样从对象副本的生成到事务处理完毕的过程中即使抛出异常也不影响对象的状态。

swap 过程不可抛出异常也是一个规则，参见 ID_throwInSwap。

相关

ID_resourceLeak
ID_ownerlessResource
ID_throwInSwap

参考

Effective C++ item 29

R7.2 异常类的构造函数与异常信息相关的函数不应抛出异常

ID_exceptionInException　　exception warning

抛出异常时，或获取异常相关的信息时，如果再抛出异常不利于异常的处理与定位。

示例：

```
class MyException {
    string msg;

public:
    MyException(const char* s) {
        if (!s) {
            throw AnotherException();  // Non-compliant
        }
        msg.assign(s);
    }

    const char* what() const {
        if (msg.empty()) {
            throw AnotherException();  // Non-compliant
        }
        return msg.c_str();
    }
};
```

例中在构造函数和 what 函数中抛出异常是不符合要求的，而且要注意 string 类型的构造函数需要动态内存分配，当分配失败时也会抛出异常，有高可靠性要求的软件系统需要规避。

自定义的异常类应从标准库定义的相关异常基类派生，异常类的成员应尽量简单，如：

```
class MyException: public std::exception {
public:
    const char* what() const noexcept override {
        return "message";
    }
};
```

由标准异常类的定义，MyException 的构造函数可以保证不会抛出异常。

依据

ISO/IEC 14882:2011 18.8.1

R7.3 析构函数不可抛出异常

ID_throwInDestuctor　　exception error

析构函数抛出异常是违反异常处理机制的。

当抛出异常时，从异常被抛出到异常被处理之间的对象，也就是从“throw”到“catch”各层调用栈中的对象会被自动析构，如果在这个过程中某个对象的析构函数又抛出了异常，将引发混乱。标准规定，这种情况会直接引发 std::terminate 函数的执行，所以从析构函数抛出的异常有可能是无法被捕获和处理的。

示例（设 E0 和 E1 是不相关的异常类）：

```
class A {
    ....
public:
   ~A() try {
        if (cond0) {
            throw E0();    // Non-compliant, propagate out
        }
        else if (cond1) {
            throw E1();    // OK, digested by itself
        }
    }
    catch (const E1&) {
        ....
    }
};
```

建议将析构函数声明为 noexcept。

与析构相关的过程也不应抛出异常：

- 资源回收
- delete、delete[] 运算符
- 具有 free、clear、release 等语义的函数

另外，具有 swap、hash 等语义以及移动构造或赋值相关的过程也不应抛出异常，详见相关规则。

相关

ID_throwInHash
ID_throwInSwap
ID_throwInMove

依据

ISO/IEC 14882:2003 15.2(3)
ISO/IEC 14882:2011 15.2(3)
ISO/IEC 14882:2011 3.7.4.2-undefined

参考

C++ Core Guidelines C.36
C++ Core Guidelines C.37
C++ Core Guidelines E.16
SEI CERT DCL57-CPP
MISRA C++ 2008 15-5-1

R7.4 与 STL 标准库相关的 hash 过程不应抛出异常

ID_throwInHash　　exception suggestion

对象的 hash 过程中不应抛出异常，否则相关的容器和算法无法正常工作。

示例：

```
template <>
struct std::hash<MyType> {
    using result_type = size_t;
    using argument_type = MyType;

    size_t operator()(const MyType& x) const {
        if (!x.ptr) {
            throw exception();  // Non-compliant
        }
        return hash<size_t>()((size_t)x.ptr);
    }
};
```

标准库规定容器的 find、count 等方法应通过返回值表示对象存在与否，然而如果 hash 过程抛出异常，这些方法也会抛出异常，相当于打破了这种约定，易造成意料之外的结果。

参考

C++ Core Guidelines C.89

R7.5 对象的 swap 过程不可抛出异常

ID_throwInSwap　　exception warning

两个对象在 swap（交换）过程中，每个对象的状态都是不完整的，如果在交换中途抛出异常，对象将处于错误的状态无法恢复。

标准库中存在大量与 swap 相关的接口和算法，如果 swap 抛出异常，也会使标准库无法按约定工作，所有 swap 函数均应标记为 noexcept。

注意，swap 是保证异常安全的重要手段，不抛出异常是基本要求，详见 ID_exceptionUnsafe。

示例：

```
struct T {
    int* ptr = nullptr;

    void swap(T& a) {
        int* tmp = ptr;
        ptr = a.ptr;
        if (!ptr) {
            throw exception();  // Non-compliant
        }
        a.ptr = tmp;
    }

   ~T() {
        delete[] p;  // Problems
    }
};
```

相关

ID_exceptionUnsafe
ID_throwInMove

参考

C++ Core Guidelines C.84
C++ Core Guidelines C.85

R7.6 移动构造函数和移动赋值运算符不可抛出异常

ID_throwInMove　　exception warning

移动构造函数和移动赋值运算符在本质上相当于将当前对象与临时对象“交换”，交换过程不可抛出异常，可参见 ID_throwInSwap。

示例：

```
class T {
    ....

public:
    T(T&& a) noexcept {
        this->swap(a);
    }

    T& operator = (T&& a) noexcept {
        this->swap(a);
        return *this;
    }

    void swap(T& a) noexcept {  // Do not throw exceptions
        ....
    }
};
```

如能保证成员被正确初始化，可采用例中模式有效实现各种移动接口。

相关

ID_throwInSwap

参考

C++ Core Guidelines C.66

R7.7 禁用含 throw 关键字的异常规格说明

ID_forbidThrowSpecification　　exception warning

由 throw 关键字声明的动态异常规格说明已过时，应采用由 noexcept 关键字声明的方式。

将所有可能抛出的异常详细列出，尤其是牵扯到第三方不可控代码时，会增大代码的管理成本，而且各编译器相关的实现方式并未统一，现已移出标准。

示例：

```
int foo() throw(Exception);  // Non-compliant
```

应改为：

```
int foo() noexcept(false);   // Compliant
int bar() noexcept;          // Compliant
```

例外：

```
int bar() throw();           // Let it go?
```

空的 throw 异常规格说明与 noexcept 等价，是一种惯用写法，可根据配置项放宽要求，本规则不建议使用 throw 关键字。

配置

forbidEmptyThrowSpecification：为 true 时报出空 throw 异常规格说明，否则放过

依据

ISO/IEC 14882:2011 D.4-deprecated
ISO/IEC 14882:2017 D.3-deprecated

参考

C++ Core Guidelines E.12
C++ Core Guidelines E.30

R7.8 重新抛出异常时应使用空 throw 表达式（throw;）

ID_improperRethrow　　exception warning

重新抛出异常时应使用空 throw 表达式，避免异常对象的精度损失或不必要的复制。

示例：

```
class EBase {};
class EDerive: public EBase {};

void foo() {
    try {
        throw EDerive();
    }
    catch (const EBase& e) {
        throw e;  // Non-compliant, use throw;
    }
```

```
}

void bar() {
    try {
        foo();
    }
    catch (const EDerive& e) {
        // Cannot catch EDerive
    }
}
```

注意，例中 foo 函数虽然捕获的是 EDerive 对象，但 throw e; 抛出的是 EBase 对象，这也是一种“对象切片”问题，造成了对象类型的“精度损失”。将 throw e; 改为 throw; 可解决这种问题。

依据

ISO/IEC 14882:2003 15.1(6)
ISO/IEC 14882:2011 15.1(8)

R7.9 不应在 catch 块外使用空 throw 表达式（throw;）

ID_rethrowOutOfCatch　　exception warning

空 throw 表达式用于重新抛出当前捕获的异常，用在 catch 块外是危险的，增大了流程控制的复杂性。

如果当前没有捕获异常的话，空 throw 表达式会引发 std::terminate 函数的执行，这种情况下是否清理调用栈之间的对象则是由实现定义的。

示例：

```
void foo() {
    throw;   // Non-compliant
}

void bar() {
    try {
        throw;  // Non-compliant
    }
    catch (...) {
        // Cannot catch "throw;"
    }
}
```

依据

ISO/IEC 14882:2003 15.1(6 8)
ISO/IEC 14882:2003 15.3(9)-implementation
ISO/IEC 14882:2011 15.1(8 9)
ISO/IEC 14882:2011 15.3(9)-implementation

参考

MISRA C++ 2008 15-1-3

R7.10 不应抛出过于宽泛的异常

ID_throwGenericException　　exception warning

抛出过于宽泛的异常如 std::exception、std::logic_error、std::runtime_error 等，使异常处理失去针对性，无法做到具体问题具体处理，而且处理这种异常时很可能将本不应处理的异常一并捕获，造成混乱。

示例：

```
int foo(int a) {
    if (a < 0) {
        throw std::exception();  // Non-compliant
    }
    return bar(a);  // Other exceptions may be thrown
}

void baz(int a) {
    try {
        foo(a);
    } catch (std::exception& e) {  // Other exceptions are also caught
        ....
    }
}
```

foo 函数在参数不符合要求时抛出 std::exception 类的异常，过于宽泛，如果 bar 函数抛出其他异常，也会被当作"参数不符合要求"处理，这显然是错误的。

正确的做法是为每种异常定义明确的子类：

```
class WrongArg: public std::exception {
public:
    WrongArg() {}
};

int foo(int a) {
    if (a < 0) {
        throw WrongArg();  // Compliant
    }
    return bar(a);  // Other exceptions may be thrown
```

```
}

void baz(int a) {
    try {
        foo(a);
    } catch (WrongArg& e) {  // Right
        ....
    }
}
```

相关

ID_catch_generic

参考

CWE-397

R7.11 不应抛出非异常类型的对象

ID_throwNonExceptionType　　exception warning

字符串或普通变量以及非异常相关的对象不应被当作异常抛出，否则意味着异常相关的设计是不健全的。

完善的异常处理机制应包含如下几点：

- 可提供对异常情况的准确描述
- 可将异常情况有效分类
- 可方便地处理异常并进行调试

值得强调的是 throw、try、catch 等关键字应专注于异常处理，不应使用这些关键字控制程序的业务流程，业务代码与异常处理代码应有明显区别，否则会使代码含混不明，效率也会降低。

示例（选自 C++ Core Guidelines）：

```
// don't: exception not used for error handling
int find_index(vector<string>& vec, const string& x)
{
    try {
        for (gsl::index i = 0; i < vec.size(); ++i)
            if (vec[i] == x) throw i;  // found x
    }
    catch (int i) {
        return i;
    }
    return -1;   // not found
}
```

这段代码比常规的实现要慢很多。

相关

ID_catch_nonExceptionType

参考

C++ Core Guidelines E.14
C++ Core Guidelines E.3

R7.12 不应将指针作为异常抛出

ID_throwPointer　　exception suggestion

如果将指针作为异常抛出，并且该指针指向动态创建的对象，会增加不必要的内存管理开销，也容易造成意料之外的错误。

示例：

```
class E { .... };

void foo(int i) {
    static E e1;
    E* e2 = new E;
    if (i > 0) {
        throw &e1;  // Non-compliant
    } else {
        throw e2;   // Non-compliant
    }
}

void bar(int i) {
    try {
        foo(i);
    } catch (E* e) {
        // ‘e’ should be deleted or not??
    }
}
```

例中对捕获的异常指针不论释放还是不释放都有问题，改为抛出对象的方式可有效避免这种问题。

参考

MISRA C++ 2008 15-0-2

R7.13 不应抛出 NULL

ID_throwNULL　　exception warning

在 C++ 语言中，虽然 NULL 表示空指针，然而在多数环境中 throw NULL 相当于 throw 0，类型的不明确会造成对异常的错误捕捉。

示例：

```
void foo() {
    throw NULL;  // Non-compliant
}

void bar() {
    try {
        foo();
    } catch (int) {  // Which handler?
        ....
    } catch (int*) {
        ....
    }
}
```

本例意在抛出一个空指针，然而会被捕获整数的 catch 块捕获。

相关

ID_deprecatedNULL
ID_throwNonExceptionType
ID_throwPointer

依据

ISO/IEC 14882:2003 C.2.2.3(1)-implementation
ISO/IEC 14882:2011 C.3.2.4(1)-implementation
ISO/IEC 14882:2017 C.5.2.7(1)-implementation

参考

CWE-351
MISRA C++ 2008 15-1-2

R7.14 不应抛出 nullptr

ID_throwNullptr　　exception warning

---

nullptr 可被所有接受指针的 catch 块捕捉，使异常处理失去针对性，故不应抛出 nullptr。

示例：

```
void foo() {
    throw nullptr;  // Non-compliant
}

void bar() {
    try {
        foo();
    } catch (int*) {  // Which handler?
        ....
    } catch (char*) {
        ....
    }
}
```

相关

ID_throwPointer

参考

MISRA C++ 2008 15-0-2

R7.15 禁用 C++ 异常

ID_forbidException　　exception warning

本规则适用如下场景，可酌情选取。

1. 对时空性能有严格要求的项目
2. 代码所属框架不支持异常处理
3. 对 C 语言高度兼容的代码
4. 项目没有依照异常安全的理念实施

利用返回值或错误码的错误处理方式要求检查可能产生错误的每一个步骤，有些出错情况可能被遗漏，C++ 的异常机制可大幅简化这种繁琐的方式，使代码更专注于事务或算法的实现，而且 C++ 异常是不可被忽略的，然而 C++ 的异常机制是需要一定开销的，对代码的设计与实现也有更严格的要求。

如果异常情况频繁出现，其成本是不可被忽视的，不适用于具有高性能要求的实时软件系统。如果代码所属项目没有依照异常安全的理念实施，使用异常反而会造成更多问题，可参见 ID_exceptionUnsafe 的进一步讨论。

相关

ID_exceptionUnsafe

参考

C++ Core Guidelines E.6
Google C++ Style Guide.Other C++ Features.Exceptions

8. Function

R8.1 main 函数的返回类型只应为 int

ID_mainReturnsNonInt　　function warning

---

main 函数的返回值可作为整个进程执行情况的总结，按惯例返回 0 或 EXIT_SUCCESS 表示执行成功，非 0 或 EXIT_FAILURE 表示执行失败，main 函数的返回值会作为标准 exit 函数的参数。

应采用标准明确支持的方式：

```
int main(void) { .... }  // Compliant
int main(int argc, char *argv[]) { .... }  // Compliant
```

如果将返回值设为 void 或其他非 int 类型，均不合规。

```
void main() { .... }  // Non-compliant
bool main() { .... }  // Non-compliant
```

依据

ISO/IEC 9899:2011 5.1.2.2.1(1)
ISO/IEC 9899:2011 5.1.2.2.3(1)

参考

C++ Core Guidelines F.46

R8.2 main 函数不应被重载，也不应声明为 inline、static 或 constexpr

ID_illFormedMain　　function warning

---

main 函数作为程序的入口，链接器需对其特殊处理，标准规定 main 函数不应被重载，也不应声明为 inline、static 或 constexpr。

示例：

```
static int main(const char* s) {  // Non-compliant
    ....
}

static int main(const wchar_t* w) {  // Non-compliant
    ....
}

int main()
{
    main(L"abc");  // Non-compliant
}
```

那么在 C++ 语言中可以用 noexcept 修饰 main 函数吗？标准没有明确规定，本规则也不建议这样做，不论是否使用 noexcept，在 main 函数中抛出异常都会引起 std::terminate 的执行，而且这样做也不符合惯例。

依据

ISO/IEC 14882:2003 3.6.1(2 3)
ISO/IEC 14882:2011 3.6.1(2 3)
ISO/IEC 14882:2017 6.6.1(2 3)

R8.3 函数不应在头文件中实现

ID_definedInHeader　　function warning

在头文件中实现的函数，如果不是内联、静态或模板函数，则可能被引入不同的翻译单元（translate-unit）造成编译冲突。

头文件也是项目文档的重要组成部分，头文件的主要内容应是类型或接口的声明，有必要保持头文件简洁清晰。除非函数很简短，否则不建议在头文件中内联实现，大段的函数实现会影响头文件的可读性。

示例：

```
// In a header file
int foo() {          // Non-compliant, add ‘inline’ or move it to a cpp file
    return 1;
}

inline int bar() {   // Compliant
    return 2;
}
```

对于较为复杂的模版函数，建议将其实现与主体头文件分离，如：

```
// In B.h
template <class T>
struct B {
    T foo(T&);
};

#include "impl/B.inc"

// In impl/B.inc
template <class T>
T B<T>::foo(T& p) {
    ....             // Complex implementation
}
```

参考

C++ Core Guidelines SF.2

R8.4 函数的参数名称在声明和实现处应保持一致

ID_inconsistentParamName　　function warning

函数的参数名称在声明和实现处不一致甚至顺序也不相同，会对函数的调用者造成误导，而且不能排除是实现上的错误。

示例：

```
int foo(int a, int b);  // Prototype

int foo(int b, int a) {  // Non-compliant, which is which??
    return a? b + 1: 0;
}
```

参考

MISRA C++ 2008 8-4-2

R8.5 多态类的对象作为参数时不应采用值传递的方式

ID_paramMayBeSlicing　　function warning

将派生类对象通过传值的方式转换为基类对象后，不再遵循多态机制，易产生意料之外的错误，应采用指针或引用的方式传递多态类对象。

示例：

```
class A {
public:
    virtual int fun();
};

void foo(A);   // Non-compliant

void bar(A&);  // Compliant
void baz(A*);  // Compliant
```

相关

ID_objectSlicing

参考

C++ Core Guidelines C.145
C++ Core Guidelines ES.63

R8.6 不应存在未被使用的具名形式参数

ID_paramNotUsed　　function suggestion

如果函数的某个参数在函数内没有被用到过，意味着函数的功能与设计预期存在差距。

如果某个参数确实不需要被用到，建议尽量从函数的声明中将其删除，如果需要遵循某种约定而必须保留参数的话（如虚函数或回调函数），不妨在参数的声明中将参数的名称删掉，使该参数成为抽象声明，并佐以一定的注释说明。

有时编译器会对没有用到的参数给出警告，为了消除警告有人会采用“参数 = 参数”或“(void) 参数”的方式来消除警告，这是不可取的，如：

```
void fun(int a) {
    a = a;  // Or '(void)a', not recommended
    ....
}
```

应改为：

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

这样编译器不会给出警告，而且也不会有多余的代码。

参考

C++ Core Guidelines F.9
MISRA C 2012 2.7
MISRA C++ 2008 0-1-11

R8.7 由 const 修饰的参数应为引用或指针

ID_paramPassedByValue　　function warning

函数参数按值传递时会产生复制及构造开销，而且如果有 const 修饰，意味着对象不可改变，那么按值传递是没有意义的。

示例：

```
void fun(const string s) {  // Non-compliant
    ....
}
```

例中参数 s 为按值传递的对象，每当 fun 函数被调用时，s 都会作为一个新的对象被构造，因为其值又不能被改变，所以这种构造是没有意义的，利用常量引用即可解决这个问题：

```
void fun(const string& s) {  // Compliant
    ....
}
```

改为常量引用后，s 的值和原来一样不可被改变，而且不需要额外的传值开销。

参考

C++ Core Guidelines F.16

R8.8 转发引用只应作为 std::forward 的参数

ID_illForwardingReference　　function warning

不应混淆转发引用与右值引用，除作为 std::forward 的参数之外，不应对转发引用再有任何操作。

转发引用是类型为 T&& 的参数，T 为函数模板类型，无论左值还是右值均可被这种参数接受，而且 const、volatile 等属性也会被忽略，由于含有不确定的状态，所以直接操作转发引用是不妥的，只应通过 std::forward 交由合适的接口处理。

示例：

```
template <class T>
void bar(T&& x) {
    x.foo();        // Non-compliant
}
```

例中参数 x 是转发引用，并不是一般的右值引用，在 bar 函数内部并不知道 x 是左值还是右值，而且 x 对应的实际参数也可能被 const 或 volatile 修饰，所以直接调用 x 的 foo 成员会引发逻辑上的混乱。

相关

ID_unsuitableForward

参考

C++ Core Guidelines F.19

R8.9 局部变量在使用前必须初始化

ID_localInitialization　　function error

未经初始化即使用的局部变量，其值是不确定的，意味着程序存在严重逻辑错误。

示例：

```
int foo() {
    int a;
    if (cond) {
        a = 0;
    }
    return a;  // Non-compliant, an indeterminate value
}
```

例中变量 a 的初始化依赖条件 cond，在条件范围之外使用是错误的。
建议变量在声明处初始化，即使不方便在声明处初始化，也应该在声明的附近进行不依赖条件的初始化。

建议的模式：

```
int a = 0;  // Compliant

int b;
b = foo();  // Compliant
```

不建议的模式：

```
int a;
if (some_cond) {
    a = foo();
}
....
if (another_cond) {
    use(a);
}
```

a 的初始化依赖于条件 some_cond，即使 some_cond 和 another_cond 有一定相关性可以保证对 a 的读取是正确的，也会造成潜在的维护困难，当条件比较复杂时极易出错。

依据

ISO/IEC 14882:2003 8.5(9)
ISO/IEC 14882:2011 8.5(11)
ISO/IEC 14882:2017 11.6(12)

参考

CWE-909
CWE-908
CWE-824
CWE-457
C++ Core Guidelines ES.20
MISRA C 2004 9.1
MISRA C 2012 9.1
MISRA C++ 2008 8-5-1

R8.10 成员须在声明处或构造时初始化

ID_memberInitialization　　function warning

由于成员声明的位置和使用的位置相距较远，所以更容易造成未初始化先使用的问题，应该在声明处或构造函数中初始化所有需要初始化的成员。

示例：

```
struct A {
    int x;
    int y = 0;
    int z;

    A(int i): x(i) {  // Non-compliant, Missing initialization of ‘z’
    }
};
```

例中构造函数没有对 z 初始化是不符合要求的，尤其是公有成员出现这种问题时会造成更大的风险，建议所有成员都在声明处初始化。

应改为：

```
struct A {
    int x = 0;  // Good
    int y = 0;  // Good
    int z = 0;  // Good

    A(int i): x(i) {  // Compliant
    }
};
```

参考

CWE-908
CWE-824
C++ Core Guidelines C.41

R8.11 基类对象构造完毕之前不可调用成员函数

ID_illMemberCall　　function warning

基类对象未构造完毕时调用成员函数会导致标准未定义的错误。

示例：

```
struct A {
    A(int);
};

struct B: A {
    B(): A(member()) {  // Non-compliant, undefined behavior
    }

    int member();
};
```

例中成员函数的返回值作为基类构造函数的参数，而这时基类对象尚未构造，相当于成员函数的调用者没有被初始化，这是一种逻辑错误。

依据

ISO/IEC 14882:2011 12.6.2(13)-undefined

R8.12 在面向构造或析构函数体的 catch 块中不可访问非静态成员

ID_illMemberAccess　　function error

当流程进入面向构造或析构函数体的 catch 块时，非静态成员的生命周期经结束，如果继续访问会导致标准未定义的问题。

示例：

```
class A {
    int i = 0;

public:
    A() try {        // Function-try-block
        ....
    } catch (...) {  // A handler of the function-try-block
        access(i);   // Non-compliant, ‘i’ may no longer exist
    }

   ~A() try {
        ....
    } catch (...) {
        access(i);   // Non-compliant, ‘i’ may no longer exist
    }
};
```

例中 access(i) 等访问是有问题的。

应调整实现或将“function-try-block”改为普通“try-block”：

```
A::A() {
    try {
        ....
    } catch (...) {
        access(i);   // Compliant
    }
}
```

依据

ISO/IEC 14882:2011 15.3(10)-undefined

参考

MISRA C++ 2008 15-3-3

R8.13 成员初始化应遵循声明的顺序

ID_disorderedInitialization　　function error

类成员的初始化顺序是按声明的顺序进行的，如果用后面的成员初始化前面的成员，就会造成错误。

示例：

```
struct T {
    int* p;
    size_t n;

    T(size_t s): n(s), p(new int[n])  // Non-compliant
    {}
};
```

虽然在初始化列表中 n 在 p 的前面，但实际上 n 仍然在 p 之后被初始化，“new int[n]”会造成严重错误。

应改为：

```
struct T {
    size_t n;
    int* p;

    T(size_t s): n(s), p(new int[n])  // Compliant
    {}
};
```

调整了 n 和 p 的声明顺序，使 n 先于 p 初始化即可解决问题。

几个特例：

```
struct A {
    int*& a;
    int*  b;
    int*  c;
    int   d[123];

    A(int): a(b), b(c), c(d) {}
};
```

a 为引用， b 的地址在初始化之前就确定了，所以“a(b)”没问题
b 为指针，用 c 的值初始化 b 的值是不对的
d 为数组，也是一个地址，所以“c(d)”没有问题

依据

ISO/IEC 14882:2003 12.6.2(5)
ISO/IEC 14882:2011 12.6.2(10)

参考

C++ Core Guidelines C.47

R8.14 在构造函数中不应调用虚函数

ID_virtualCallInConstructor　　function warning

虚函数在构造函数中的多态性不生效，而且调用未定义的纯虚函数会导致标准未定义的错误。

示例：

```
class A {
    int a;
    virtual int foo() { return 0; }

public:
    A(): a(foo()) {}  // Non-compliant
};

class B: public A {
    int foo() override { return 1; }
};
```

虽然 B 重写了 foo 函数，但在 A 的构造函数中不生效，成员 a 的值总为 0，这往往意味着错误，也会使维护者产生相当大的误解。

参考

C++ Core Guidelines C.82
Effective C++ item 9

R8.15 在析构函数中不应调用虚函数

ID_virtualCallInDestuctor　　function warning

虚函数在析构函数中的多态性不生效，而且调用未定义的纯虚函数会导致标准未定义的错误。

示例：

```
class A {
    virtual void clear() {}

public:
    A() {}
   ~A() { clear(); }  // Non-compliant
};

class B: public A {
    int* p = new int[8];
    void clear() override { delete[] p; }
};
```

虽然 B 重写了 clear 函数，但在 A 的析构函数中不生效，相关内存没有被正确释放。

应将基类的析构函数设为虚函数，在派生类的析构函数中释放资源：

```
class A {
    ....
    virtual ~A() {}
};

class B: public A {
    ....
   ~B() { delete[] p; }  // Compliant
};
```

参考

C++ Core Guidelines C.82
Effective C++ item 9

R8.16 拷贝构造函数应避免实现复制之外的功能

ID_sideEffectCopyConstructor　　function warning

拷贝构造函数可以被优化而不被执行，而且这种优化是由实现定义的。

拷贝构造函数只应专注于复制对象，避免复制之外的功能，如修改静态成员或对环境产生影响。

示例：

```
class A {
    int i;
    static int s;

public:
    A(): i(0) {}
    A(const A& rhs) {
        i = a.i;     // Compliant
        s++;         // Non-compliant
        ++i;         // Non-compliant
    }
};

A foo() {
    return A();
}

void bar() {
    A a = foo();
    ....
}
```

bar 函数用 foo 函数返回的对象初始化 a 对象，理论上应执行拷贝构造函数，但标准允许编译器将 foo 函数返回的临时对象直接作为 a 对象，这种优化称为“copy elision”，拷贝构造函数被执行的次数可能与预期不符，所以拷贝构造函数不应存在超出复制的副作用。

进一步可参见“RVO（Return Value Optimization）”的相关介绍。

依据

ISO/IEC 14882:2003 12.8(15)-implementation
ISO/IEC 14882:2011 12.8(31)-implementation

参考

MISRA C++ 2008 12-8-1

R8.17 赋值运算符应妥善处理参数就是自身对象时的情况

ID_this_selfJudgement function warning

赋值运算符应妥善处理参数就是自身对象时的情况，防止资源分配或回收的冲突。

示例：

```
A a;
a = a;  // Self assignment ...
```

如果例中 A 是需要深拷贝的类，在赋值运算符中应判断这种对自身赋值的情况：

```
A& A::operator = (const A& rhs) {
    if (this != &rhs) {  // Required
        ....
    }
    return *this;
}
```

如果 A 的拷贝构造函数和交换方法齐备，也可按"复制 - 交换"模式实现：

```
A& A::operator = (const A& rhs) {
    A tmp(rhs);
    this->swap(tmp);   // Good
    return *this;
}
```

利用创建临时对象并与之交换的方法，也有效规避了冲突，这种方法使各函数更专注于自己的职责，不必重复编写分配和回收相关的代码，建议采用这种方法。

参考

C++ Core Guidelines C.62

R8.18 避免无效写入

ID_invalidWrite function warning

对于内存中的数据，写入之后应被读取，如果出现：

1. 写入后未经读取再次被无条件写入
2. 写入后未经读取即结束了函数的执行

这种写入是无效的，出现这种问题往往意味着逻辑错误或功能不完整。

对象在初始化时的写入和 volatile 型数据可不受本规则限制。

示例：

```
void foo(int& a, int& b) {
    a = 123;   // Non-compliant
    a = 456;
}
```

例中参数 a 被赋值为 123 之后，无条件地又被赋值为 456，显然第一次赋值是没有意义的，很有可能是漏掉了什么。

又如：

```
int bar() {
    int i = baz();
    return i++;    // Non-compliant
}
```

例中 bar 函数返回变量 i 自增前的值，自增运算是没有意义的。

对象的初始化可不受本规则限制，如：

```
int baz() {
    int n = 0;    // OK
    if (cond) {
        n = 123;
    } else {
        n = 456;
    }
    return n;
}
```

例中局部变量 n 初始化后经由 if-else 分枝，在其两个分枝中都被赋值，也相当于被无条件写入，但在声明处初始化是值得提倡的，故这种情况不受本规则限制。

R8.19 不应存在得不到执行机会的代码

ID_unreachableCode　　function error

得不到执行机会的代码是没有意义的，往往意味着逻辑错误。

这种代码的成因主要有：

1. 所在函数无法被调用
2. 之前的所有分枝都提前结束了函数的执行
3. 之前的必经分枝中存在不会结束执行的代码
4. 所在分枝的条件恒为假
5. 所在分枝被其他分枝遮盖

第 1 点特化为：ID_staticNotUsed、ID_privateNotUsed
第 4 点特化为：ID_constLogicExpression、ID_invalidCondition、ID_switch_caseOutOfRange
第 5 点特化为：ID_if_identicalCondition、ID_if_hiddenCondition

示例：

```
int fun() {
    if (cond) {
        return 0;
    } else {
        return 1;
    }
    statements    // Non-compliant, unreachable
}
```

例中 statements 之前的所有分枝都会结束函数的执行，所以 statements 不会被执行。

例外：
在多分枝结构的末尾，统一安置一条结束函数执行的语句是一种好的编程习惯，即当 statements 只包含一条 return 或 throw 语句时可以不算作违规。

另外，在正式代码中不应存在如下形式的代码：

```
if (false) { .... }
while (false) { .... }
for (;false;) { .... }
```

也不应该在 return 语句之后存在其他语句，这种代码如果不是被人恶意篡改，就是出于某种目的将本已无效的代码遗留了下来，可参见 ID_constLogicExpression、ID_invalidCondition 的进一步讨论。

建议时刻保持代码的整洁，并将维护过程中的变动及时地保存在版本管理系统中，这样可以清晰地查看各版本之间的变动，而如果将无效代码与有效代码混在一起，势必造成维护的负担。

相关

ID_staticNotUsed
ID_privateNotUsed
ID_constLogicExpression
ID_invalidCondition
ID_switch_caseOutOfRange
ID_if_identicalCondition
ID_if_hiddenCondition

参考

CWE-561
MISRA C 2004 14.1
MISRA C 2012 2.1
MISRA C++ 2008 0-1-1

R8.20 不应存在没有副作用的语句

ID_missingSideEffect　　function error

不能对程序状态产生影响的语句称为无“副作用（side effect）”的语句，往往属于笔误或调试痕迹，应当修正或去除。

示例（设 a、b、p 为变量或指针）：

```
a == b;  // Non-compliant
```

单纯的判等是没有副作用的，很可能是赋值语句的笔误。

```
*p++;    // Non-compliant
```

单纯从某个地址进行读取是没有副作用的，这是一种对运算符优先级理解不当造成的常见错误，应改为 (*p)++;

```
p->fun;  // Non-compliant
```

由变量名或无实参列表的函数名作为一个语句是没有副作用的，此语句应改为正确的函数调用。

```
+a;      // Non-compliant
```

正号是没有副作用的，此句很可能应为 ++a;

如果语句为逻辑与表达式，左子表达式可以作为右子表达式的条件，故左子表达式可以无副作用，而右子表达式一定要有副作用，如：

```
p && p->fun();  // OK
p && p->fun;    // Non-compliant
p->fun() && p;  // Non-compliant
```

如果语句为逻辑或表达式，则要求其左右子表达式均有副作用，如：

```
p || p->fun();  // Non-compliant
p || p->fun;    // Non-compliant
p->fun() || p;  // Non-compliant
```

参考

CWE-1164
CWE-482
MISRA C 2004 14.2
MISRA C 2012 2.2
MISRA C++ 2008 0-1-9

R8.21 有返回值的函数其所有分枝都应有明确的返回值

ID_notAllBranchReturn　　function error

当函数的某个分枝没有明确的返回值时会引发标准未定义的错误。

示例：

```
int fun() {
    if (condition1) {
        return 1;
    }
    else if (condition2) {
        return 2;
    }
    else if (condition3) {  // Non-compliant if no return value
    }
    else {
        return 4;
    }
}
```

当符合 condition3 的条件时，fun 函数的调用者将得到一个错误的返回值。

例外：
在 main 函数的结尾如果不显式调用 return 语句，编译器会自动添加 return 0，故对 main 函数不作要求。

依据

ISO/IEC 9899:1999 6.9.1(12)-undefined
ISO/IEC 9899:1999 5.1.2.2.3(1)
ISO/IEC 9899:2011 6.9.1(12)-undefined
ISO/IEC 9899:2011 5.1.2.2.3(1)
ISO/IEC 14882:2011 6.6.3(2)-undefined
ISO/IEC 14882:2017 9.6.3(2)-undefined

参考

CWE-394
MISRA C 2004 16.8
MISRA C 2012 17.4
MISRA C++ 2008 8-4-3

R8.22 不可返回局部对象的地址或引用

ID_localAddressFlowOut　　function error

局部对象的生命周期在函数返回后结束，其地址或引用也会失效，如果继续访问会造成标准未定义的错误。

示例：

```
int* foo() {
    int i = 0;
    ....
    return &i;  // Non-compliant
}

int& bar() {
    int i = 0;
    ....
    return i;   // Non-compliant
}

int&& baz() {
    int i = 0;
    ....
    return std::move(i);   // Non-compliant
}
```

依据

ISO/IEC 9899:1999 6.2.4(2)-undefined
ISO/IEC 9899:2011 6.2.4(2)-undefined

参考

CWE-562
C++ Core Guidelines F.43

R8.23 合理设置 lambda 表达式对变量的捕获方式

ID_unsuitableCapture　　function warning

如果 lambda 表达式只在函数内部使用，可采用捕获引用的方式；如果 lambda 表达式可以超出函数作用域，应采用捕获值的方式。

示例：

```
auto foo() -> function<int()> {
    int i = 0;
    ....
    return [&]() { return ++i; };  // Non-compliant
}
```

例中的 lambda 表达式引用了局部变量 i，但返回后 i 的地址不再有效，会引发标准未定义的错误。

另外，要注意解引用指针造成的间接引用：

```
class A {
    int i;

public:
    auto bar() {
        return [=]() { return i; };  // Bad
    }
};
```

例中的 lambda 表达式通过值捕获变量，this 指针也被捕获，成员变量 i 是通过 this 指针的隐式解引用获取到的，如果 lambda 表达式在 this 指针的生命周期之外执行，就会造成错误。

应改为：

```
auto A::bar() {
    return [*this]() { return i; };  // OK
}
```

如果需要捕获 this 指针，则应显式捕获所有相关变量，避免使用“[=]”。

相关

ID_localAddressFlowOut

依据

ISO/IEC 14882:2011 5.1.2(7)

参考

C++ Core Guidelines F.52
C++ Core Guidelines F.53
C++ Core Guidelines F.54

R8.24 函数不应返回右值引用

ID_returnRValueReference　　function suggestion

函数返回右值引用的实际价值有限，而且很容易产生错误。

示例：

```
class A { .... };

A&& foo() {          // Non-compliant
    return A();
}

void bar() {
    A&& a = foo();
    access(a);       // Undefined behavior, ‘a’ refers to an invalid object
}
```

例中 foo 函数返回类型为右值引用，这种情况下返回临时对象一定是错误的，临时对象在返回前析构，返回的是无效引用。

也不应返回局部对象的右值引用，如：

```
A&& baz() {  // Non-compliant
    A a;
    ....
    return std::move(a);
}
```

和返回临时对象一样，对象 a 在函数返回前析构，返回的也是无效引用。

应直接返回对象，而不是对象的右值引用：

```
A foo() {  // Compliant
    return A();
}

A baz() {  // Compliant
    A a;
    ....
    return a;
}
```

对于函数引用的参数，或函数作用域之外的对象，如果通过 move 返回右值引用，如：

```
A&& baz(A& a) {  // Non-compliant
    do_something_to(a);
    return std::move(a);
}
```

这种情况在运行机制上可能没有问题，但满足的实际需求较为有限，而且相当于将 do_something_to(a) 和 move(a) 两种事务合在一个函数中，在某种程度上违反了“单一职责原则”。

综上所述，应统一要求函数不应返回右值引用。

相关

ID_localAddressFlowOut

参考

C++ Core Guidelines F.45

R8.25 函数返回值不应为 const 对象

ID_returnConstObject　　function suggestion

函数返回 const 对象不利于移动构造或移动赋值等机制，也可能本意是返回引用，但遗漏了引用符号。

示例：

```
const vector<int> fun() {  // Non-compliant
    return { 1, 2, 3 };
}

vector<int> obj(fun());    // Call ‘vector(const vector&)’
```

fun 返回 const 对象，构造 obj 对象时只能进行深拷贝，无法利用移动构造等特性。

应改为：

```
vector<int> fun() {        // Compliant
    return { 1, 2, 3 };
}

vector<int> obj(fun());    // Call ‘vector(vector&&)’, more efficient
```

这样可以利用移动构造函数提高效率。

对于遵循 C++11 之前标准的代码，也不应返回 const 对象，函数返回的对象本来就需要通过 const 引用或传值的方式被后续代码使用，所以将返回值设为 const 的意义不大。

参考

C++ Core Guidelines F.20

R8.26 返回值应与函数的返回类型相符

ID_returnOdd　　function warning

返回值与返回类型不符的情况：

- 返回类型为 bool 型，却返回了非 true 非 false、非 0 非 1 的常量
- 返回类型为指针，却返回了非 0、非 NULL、非 nullptr 等常量
- 返回类型为整数，却返回了 NULL、true、false 等常量

这些问题可能是在维护过程中产生的，也可能意味着逻辑错误，而且很容易造成误导，需谨慎对待。

示例：

```
bool foo() { return NULL; }   // Non-compliant
int  bar() { return false; }  // Non-compliant
int* baz() { return '\0'; }   // Non-compliant
```

参考

MISRA C++ 2008 4-10-1

R8.27 函数返回值不应为相同的常量

ID_returnSameConst　　function warning

如果一个函数有多个返回语句，但所有返回值都是相同的常量，那么这个函数的返回值是没有意义的。

示例：

```
bool foo(int a) {
    if (a > 100) {
        return true;
    }
    if (a > 50) {
        return true;
    }
    return true;   // Non-compliant, all return values are the same
}
```

例外：
当函数可以抛出异常时不受本规则限制。

R8.28 基本类型的返回值不应使用 const 修饰

ID_returnSuperfluousConst　　function warning

基本类型（char、int、long 等）的返回值本来就不能作为可修改的左值，const 修饰符是多余的。

示例：

```
const int foo() {  // Non-compliant, ‘const’ is superfluous
    return 123;
}

class A {
    int a = 123;

public:
    int& foo() { return a;}
    const int foo() const { return a; }  // Non-compliant, missing ‘&’
};
```

出现这种问题说明设计与使用存在一定的偏差，也可能本意是返回引用，而书写时漏掉了引用符号造成的。

依据

ISO/IEC 14882:2003 3.10(5)
ISO/IEC 14882:2011 3.10(1)

R8.29 属性为 noreturn 的函数中不应出现 return 语句

ID_unsuitableReturn　　function warning

属性为 noreturn 的函数中出现 return 语句说明函数还是可以正常返回的，这种矛盾会对调用者造成很大困扰，而且会导致标准未定义的问题。

示例：

```
[[noreturn]] void foo(int a) {
    if (a < 0) {
        return;  // Non-compliant, undefined behavior
    }
    abort();
}
```

依据

ISO/IEC 14882:2011 7.6.3(2)-undefined

R8.30 属性为 noreturn 的函数返回类型只应为 void

ID_unsuitableReturnType function warning

属性为 noreturn 的函数返回类型不是 void 说明函数还是有返回值的，这种矛盾会对调用者造成很大困扰。

示例：

```
[[noreturn]] int foo();   // Non-compliant
[[noreturn]] void bar();  // Ccompliant
```

依据

ISO/IEC 14882:2011 7.6.3(2)-undefined

R8.31 不应出现多余的跳转语句

ID_redundantJump function warning

无返回值函数的最后一条 return 语句、循环体的最后一条 continue 语句、goto 到下一条语句等是多余的，应当去除。

示例：

```
void foo() {
    ....
    return;   // Redundant
}

void bar() {
    while (condition) {
        ....
        continue;    // Redundant
    }
}

void baz() {
    goto L;   // Redundant
L:
    ....
```

```
}
```

R8.32 va_start 或 va_copy 应配合 va_end 使用

ID_incompleteVAMacros　　function warning

可变参数列表相关的 va_start 或 va_copy 和 va_end 应在同一函数中使用，否则会导致标准未定义的错误。

示例：

```
void foo(const char* s, ...) {  // Non-compliant, missing ‘va_end(vl);’
    va_list vl;
    va_start(vl, s);
    for (const char* p = s; *p; p++) {
        ....
    }
}
```

应在返回前使用 va_end。

相关

ID_forbidVariadicFunction

依据

ISO/IEC 9899:2011 7.16.1.3(2)-undefined

R8.33 函数模版不应被特化

ID_functionSpecialization　　function warning

特化的函数模板不参与重载函数的选取，不属于常规用法，且容易造成混乱。

示例：

```
template <class T>
int foo(T) {           // #1
    return 0;
}

template <class T>
```

```
int foo(T*) {          // #2
    return 1;
}

template <>
int foo<int*>(int*) {  // #3, non-compliant, specialization of #1
    return 2;
}

int main() {
    int* p = nullptr;
    cout << foo(p) << '\n';  // What is output?
}
```

输出 1，特化的函数模板不参与重载函数的选取，所以只会在 #1 和 #2 中选取，foo(p) 与 #2 更贴近，而 #3 是 #1 的特化，所以不会选取 #3，这种情况下 #3 是无效的。

应去除对函数模板的特化，改为普通重载函数：

```
int foo(int*) {   // #3, compliant, safe and brief
    return 2;
}
```

这样例中 main 函数会输出 2。

参考

C++ Core Guidelines T.144
MISRA C++ 2008 14-8-1

R8.34 函数的标签数量应在规定范围之内

ID_tooManyLabels　　function warning

标签过多意味着函数内部的跳转逻辑过于复杂，违反结构化设计理念，应适当重构。

对于 C 代码，建议一个函数只用一个标签作为函数统一出口，对于 C++ 代码，不建议使用标签。

示例：

```
void foo()
{
L0:
    ....
L1:
    // ... lots of labels require lots of gotos ...
    // ... lots of gotos make functions terrible ...
L100:
    ....
}
```

配置

maxLabelCount：标签数量上限，超过则报出

R8.35 函数的行数应在规定范围之内

ID_tooManyLines　　function warning

函数体过大违反模块化编程理念，使人难以阅读，更不便于维护，很有可能隐藏着各种错误，应适当重构。

示例：

```
int main()
{
    // ... 3000 lines ...
    // Who has the courage to read?
}
```

建议函数体不超过 60 行，以不需要拖拽滚动条就可以在屏幕上完整显示为宜。

配置

maxLineCount：行数上限，超过则报出

参考

C++ Core Guidelines F.2
C++ Core Guidelines F.3

R8.36 lambda 表达式的行数应在规定范围之内

ID_tooManyLambdaLines　　function warning

复杂的 lambda 表达式与其调用者的代码混在一起时，是难以阅读的，引入 lambda 表达式的目的应该是“化简”，否则应使用普通函数。

示例：

```
void foo()
{
    auto f0 = []() {
        // ... Many lines ...
    };
        // ... Many lambdas ...
        // ... Even lambdas nest lambdas ...
    auto f100 = []() {
        // ...
    };
    // Tut tut, this is a function, not a namespace,
    // use common functions instead
}
```

建议 lambda 表达式不超过 5 行，一个函数中不应有多个复杂的 lambda 表达式。

配置

maxLambdaLineCount：lambda 表达式行数上限，超过则报出

R8.37 函数参数的数量应在规定范围之内

ID_tooManyParams　　function warning

函数参数过多，意味着：

- 缺少合理的抽象机制：应将多而零散的参数按其内在联系封装成对象，从而方便地处理其逻辑关系，而不是简单地线性罗列
- 违反“单一职责原则”：参数越多，函数处理的事务自然越多，代码的可维护性自然越差

建议可供外部使用的全局函数、public 或 protected 成员函数的参数不超过 4 个，内部使用的 static 函数、private 成员函数的参数不超过 8 个。

示例：

```
void foo(A a, B b, C c, D d, E e) {  // Bad
    ....
}
```

当函数参数过多时，应按参数的逻辑职责进行封装。
假设 a 和 b 有直接逻辑关系，c、d、e 有直接逻辑关系，不妨将 a 和 b 封装成一个类，c、d、e 封装成一个类，在类的成员函数中实现相关功能，可更为清晰直观地保证逻辑关系的正确性。

```
class X {
    // ... Members and methods for ‘a’, ‘b’ ...
};

class Y {
    // ... Members and methods for ‘c’, ‘d’, ‘e’ ...
};

void foo(X x, Y y) {  // Good
    x.methods();
    y.methods();
}
```

配置

maxInnerFunParamCount：static 函数或 private 成员函数参数数量上限，超过则报出
maxParamCount：参数数量上限，超过则报出

参考

C++ Core Guidelines F.2
C++ Core Guidelines I.23

R8.38 不应定义过于复杂的内联函数

ID_complexInlineFunction　　function suggestion

不适合将函数声明为内联的情况：

- 行数超过指定限制
- 存在循环或异常处理语句
- 存在 switch 分枝语句
- 函数存在递归实现

建议内联函数的实现不要超过 3 个语句。

配置

maxInlineFunctionLineCount：内联函数行数上限，超过则报出

参考

C++ Core Guidelines F.5

R8.39 禁止 goto 语句向嵌套的或无包含关系的作用域跳转

ID_forbidGotoBlocks　　⛔ function warning

不同的作用域对应不同的条件约束，在不同的作用域间跳转是对约束的破坏，很容易导致逻辑混乱。

向嵌套的或无包含关系的作用域跳转是不应被允许的，如果是为了结束当前流程而在同层或向外层作用域跳转，则可被本规则允许。

示例：

```
int foo(int a) {
    if (a == 0) {
        goto L;    // Non-compliant, jumps into a block
    }
    for (int i = 0; i < a; i++) {
        for (int j = 0; j < a; j++) {
L:
            if (cond) {
                goto M;   // Compliant, jumps out of loops
            }
        }
        ....
    }
M:
    return a;
}
```

例中 goto L 从 if 语句跳入循环语句是应当被禁止的，而 goto M 用于结束循环流程，可以保留。

相关

ID_forbidGotoBack
ID_forbidGoto

参考

MISRA C 2012 15.3
MISRA C++ 2008 6-6-1

R8.40 禁止 goto 语句向前跳转

ID_forbidGotoBack ⛔ function suggestion

向先于当前 goto 语句定义的标签跳转，可读性较差，是公认的不良实现。

示例：

```
int foo() {
    int i = 0, j = 0;
L:
    j += 1;
    i += j;
    if (j > 100) {
        goto M;    // Compliant
    }
    goto L;        // Non-compliant
M:
    return i;
}
```

例中 goto M 向后跳转符合本规则要求，而 goto L 向前跳转不符合要求，应改用循环等结构性语句。

相关

ID_forbidGotoBlocks
ID_forbidGoto

参考

MISRA C 2012 15.2
MISRA C++ 2008 6-6-2

R8.41 禁用 goto 语句

ID_forbidGoto ⛔ function suggestion

历史表明，goto 语句会破坏程序的结构性规划，很容易导致逻辑混乱且不利于维护，在非自动生成的、对可读性有要求的代码中，建议禁用 goto 语句。

示例：

```
if (cond0) {
    goto L;   // Non-compliant
}
....
if (cond1) {
L:
    ....
}
```

语句的排列和作用域的嵌套描述了程序的静态结构，清晰的静态结构使人易于理解程序的行为，而 goto 语句会打破这种结构，无规律的跳转会显著降低代码的可读性，例中 goto L 会绕过第二个 if 语句的条件约束，可读性很差，应被禁止。

C 语言的流程管理较为简单，goto 语句可提供一定的灵活性，但不应作为常规实现手段，也应受一定的限制，在 C 代码中使用 goto 语句应遵循 ID_forbidGotoBlocks 和 ID_forbidGotoBack 等规则。

C++ 语言提供了更丰富的流程管理功能，在 C++ 代码中不应再使用 goto 语句。

下面给出 goto 语句的一种常用模式：

```
void foo(size_t n)
{
    int *a = NULL, *b = NULL, *c = NULL;
    a = (int*)malloc(n);
    if (!a) {
        goto E;
    }
    b = (int*)malloc(n);
    if (!b) {
        goto E;
    }
    c = (int*)malloc(n);    // Multiple resource allocation
    if (!c) {
        goto E;
    }
    ....
E:                          // Single exit point
    free(a);
    free(b);
    free(c);
}
```

在多次资源分配过程中，如果某次分配失败则需要释放已分配的资源，利用 goto 语句可实现资源的统一释放，在 C 代码中如果不用 goto 语句反而会很繁琐，所以这种模式在 C 代码中可以复用。

由于 C++ 提供容器、智能指针等更丰富的资源管理手段，所以不建议在 C++ 代码中再使用这种模式，即使标准库没有和相关资源对应的功能，也应该利用“RAII”等机制对其先封装再使用。

```
void foo(size_t n) {
    std::vector<int> a, b, c;   // Safe and brief
    ....
}
```

相关

ID_forbidGotoBlocks
ID_forbidGotoBack

参考

C++ Core Guidelines ES.76
MISRA C 2012 15.1

R8.42 禁用 setjmp、longjmp

ID_forbidLongjmp　　⛔ function warning

---

setjmp、longjmp 可以在函数间跳转，进一步破坏了结构化编程理念，非框架代码不应使用。

示例：

```
jmp_buf buf;

float div(int a, int b) {
    if (b == 0) {
        longjmp(buf, 1);     // Non-compliant
    }
    return (float)a / b;
}

int main() {
    if (setjmp(buf) == 0) {  // Non-compliant
        printf("%f\n", div(3, 0));
    } else {
        return 1;
    }
}
```

setjmp 返回 0 表示设置跳转位置成功，之后如果调用 longjmp 会跳回 setjmp 的位置，这时 setjmp 返回非 0 值，这种机制在 C 语言中可以用作异常处理，也可以实现“协程”等概念，但会使代码的可维护性显著降低，在普通的业务逻辑或算法实现中不应使用。

setjmp 与 longjmp 由 jmp_buf 参数关联，应在同一线程中使用，如果调用 longjmp 时没有对应的 setjmp，或 setjmp 所在函数已经结束执行，会导致标准未定义的行为，而且要注意 setjmp、longjmp 无法与 C++ 对象自动析构等机制兼容，极易造成意料之外的错误。

依据

ISO/IEC 9899:1999 7.13.2.1(2)-undefined
ISO/IEC 14882:2011 18.10(4)-undefined

参考

MISRA C 2004 20.7
MISRA C 2012 21.4
MISRA C++ 2008 17-0-5
C++ Core Guidelines SL.C.1

R8.43 避免递归实现

ID_recursion function warning

递归实现，如函数直接或间接地调用自身，易导致难以控制的堆栈溢出错误。

示例：

```
size_t foo(size_t n) {
    return n + foo(n - 1);  // Non-compliant
}
```

例中 foo 函数无条件地调用自身，是一种逻辑错误，导致无限的递归调用。

又如：

```
size_t bar(size_t n) {
    if (n > 1) {
        return n + bar(n - 1);  // Non-compliant
    }
    return n;
}
```

例中 bar 函数设置了递归条件，但仍是不可取的，当参数 n 较大时仍然可以造成堆栈溢出错误。

对于一般的功能，应尽量采用迭代、堆栈等非递归手段实现，对于难以使用非递归方式实现的特殊算法，应做到递归深度可控。

参考

MISRA C 2012 17.2
MISRA C++ 2008 7-5-4

R8.44 不应存在重复的函数实现

ID_functionRepetition　　function suggestion

不同的函数代码却完全相同或过于相似是不利于维护的。

参考

CWE-1041
C++ Core Guidelines ES.3

9. Control

9.1 If

R9.1.1 if 语句不应被分号隔断

ID_if_semicolon　　control error

---

if 语句不应被分号隔断。

示例：

```
if (condition);  // Non-compliant, see the semicolon
{
    ....
}
```

一个分号使整个 if 语句失效，这可能是笔误，也可能是需求变化不再需要条件判断了，在维护过程中加入了分号，形成了令人费解的残留代码，也不能排除是有人恶意篡改了代码，应立即修正。

参考

CWE-670

R9.1.2 在 if...else-if 分枝中不应有重复的条件

ID_if_identicalCondition　　control error

---

if...else-if 分枝的条件不应有重复，否则相同条件排在前面的分枝会得以执行，而排在后面的分枝得不到执行机会。

示例：

```
if (condition1) {
    branch1
}
else if (condition2) {
    branch2
}
else if (condition1) {  // Non-compliant, see the previous ‘condition1’
    branch3
}
else {
    branch4
}
```

例中 branch1 和 branch3 的条件是相同的，所以 branch3 不会被执行。

此问题为常见笔误，多数由复制粘贴造成，也可能是维护代码时对之前的逻辑不够了解造成的错误。

相关

ID_if_hiddenCondition

参考

CWE-670
CWE-561

R9.1.3 在 if...else-if 分枝中不应有被遮盖的条件

ID_if_hiddenCondition　　control error

---

if...else-if 分枝中，如果前面的条件被满足，后面的分枝就不会被执行，所以如果前面的条件是后面条件的一部分，或者前面的条件包含后面的条件，即使后面的条件可以被满足，其分枝也得不到执行机会。

示例：

```
if (condition1) {
    branch1
}
else if (condition2) {
    branch2
}
else if (condition1 && condition3) {  // Non-compliant, see the previous
‘condition1’
    branch3
}
else {
    branch4
}
```

如果 condition1 为 true，branch1 将得以执行，branch3 不会被执行，如果 condition1 为 false，branch3 还是不会被执行，称 branch3 被 condition1 遮盖了，branch3 永远不会得到执行机会。

如果前面的条件包含后面的条件，同样也会遮盖后面的条件，如：

```
if (condition1 || condition2) {
    branch1
}
else if (condition2) {  // Non-compliant, see the previous ‘condition2’
    branch2
}
else {
    branch3
}
```

同理，branch2 永远也不会被执行。

相关

ID_if_identicalCondition

参考

CWE-670
CWE-561

R9.1.4 if 分枝和 else 分枝的代码不应完全相同

ID_if_identicalBlock　　control error

---

if 分枝和 else 分枝完全相同会使条件判断失去意义，往往是由复制粘贴造成的错误。

示例：

```
if (condition) {
    branch
}
else {
    branch  // Non-compliant
}
```

例中 branch 表示完全相同的代码，需修正本应存在的差异，或去掉 if-else 结构。

相关

ID_if_identicalElseIfBlock
ID_if_identicalImplicitElseBlock

参考

CWE-670

R9.1.5 if...else-if 各分枝的代码不应完全相同

ID_if_identicalElseIfBlock control warning

内容完全相同的分枝是没有意义的，也可能是由复制粘贴造成的错误。

示例：

```
if (condition1) {
    branch1
}
else if (condition2) {
    branch2
}
else {
    branch1  // Non-compliant
}
```

例中 condition1 对应分枝和 else 分枝的内容完全相同，应该合并成一个分枝，或修正本应存在的差异：

```
if (condition2) {
    branch2
}
else {
    branch1  // Compliant
}
```

例外：
如果分枝内容较少，为了使代码更清晰可以接受适当的重复，但如果分枝内容很多就不应重复了，不妨指定一个数量限制，当重复分枝的符号数量超过这个限制时算作违规，否则放过。

配置

blockTokenCountThreshold：重复分枝符号数量限制，不检查符号数量小于该值的分枝

相关

ID_if_identicalBlock

参考

CWE-670

R9.1.6 if 分枝和其隐含的 else 分枝的代码不应完全相同

ID_if_identicalImplicitElseBlock control warning

带有 return、throw 或 break 等子句的 if 语句，其同一作用域的后续代码相当于它的 else 分枝，显然这种隐含的 else 分枝与 if 分枝完全相同是没有意义的，很可能是由复制粘贴造成的错误。

示例：

```
if (condition) {
    statements
    return;
}
statements  // Non-compliant
return;
```

例中 if 语句之内的 statements 以及 return 语句和 if 语句之外的语句完全相同，这种情况下 if 语句的条件判断是没有意义的，需修正本应存在的差异，或去掉 if 语句。

相关

ID_if_identicalElseIfBlock
ID_if_identicalSucceedingBlock

参考

CWE-670

R9.1.7 没有 else 子句的 if 语句与其后续代码相同是可疑的

ID_if_identicalSucceedingBlock control suspicious

---

if 语句与其同一作用域的后续代码完全相同是可疑的，可能是由复制粘贴造成的错误。

示例：

```
if (condition) {
    a = SOME_VALUE;
}
a = SOME_VALUE;  // Rather suspicious
```

例中对变量 a 的赋值是没有意义的。

相关

ID_if_identicalImplicitElseBlock

参考

CWE-670

R9.1.8 if 分枝和 else 分枝的起止语句不应相同

ID_if_commonStatements control suggestion

---

if 分枝和 else 分枝的起止语句如果相同，则应将其从分枝结构中提取出来，否则重复的代码不利于阅读和维护。

示例：

```
if (condition) {
    foo();
    ....
    bar();
}
else {
    foo();
    ....
    bar();
}
```

if 与 else 分枝的开头和结尾相同，应提取出来：

```
foo();
if (condition) {
    ....
}
else {
    ....
}
bar();
```

当条件分枝中的所有语句都相同时，特化为 ID_if_identicalBlock，这种情况往往意味着错误。

相关

ID_if_identicalBlock

参考

CWE-670
C++ Core Guidelines ES.3

R9.1.9 if 语句作用域的范围不应有误

ID_if_scope control warning

---

if 语句作用域的范围不应有误。

示例：

```
if (condition)
    statement1; statement2;  // Non-compliant

if (condition)
    statement1;
    statement2;  // Non-compliant
```

例中 statement2 不在 if 语句的作用域中，但看起来又和 if 语句相关，这种问题多数是由错误的宏展开或无效的缩进造成的。

为了避免这种问题，if 语句应使用大括号括起来：

```
if (condition) {
    statement1; statement2;  // Compliant
}
```

相关

ID_if_brace

参考

CWE-483

R9.1.10 如果 if 关键字前面是右大括号，if 关键字应另起一行

ID_if_mayBeElseIf　　control suspicious

当 if 关键字前面是右大括号，且 if 关键字与该大括号在同一行时，属于不良换行方式，易造成 else 等关键字的遗漏。

示例：

```
if (condition1) {
    ....
} if (condition2) {  // Rather suspicious, should it be ‘else if’??
    ....
}
```

这种情况很可能是漏掉了 else 关键字，即使没有被漏掉，也应该让 if 关键字另起一行，否则这种换行习惯会增加遗漏 else 关键字的风险，而且可读性较差。

R9.1.11 if 语句的条件不应为赋值表达式

ID_if_assignment　　control warning

虽然语言允许在 if 语句的条件中赋值，但 = 和 == 极易混淆，建议所有产生 bool 型结果的表达式，都不要包含赋值运算符。

示例：

```
HRESULT r = URLDownloadToFile(....);
if (r = S_OK) {  // Non-compliant
    ....
}
```

设 S_OK 为常量，在条件中用常量对变量赋值是没有逻辑意义的，如果条件中等号右侧为常量，基本可以判定是 == 被误写成了 =。

又如：

```
if (r = fun()) {  // Non-compliant
    ....
}
```

这也是一种公认的不良风格，应将赋值表达式拆分出来，或者在 C++ 代码中改为：

```
if (auto r = fun()) {  // Compliant
    ....
}
```

将赋值表达式加上括号，表示有意为之，是一种惯用写法：

```
if ((r = fun())) {  // Let it go?
    ....
}
```

这种情况可通过配置决定是否放过。

配置

allowEnclosedAssignment：为 true 可以放过括号括起来的赋值表达式

参考

CWE-480
CWE-481
CWE-783

R9.1.12 if 语句不应为空

ID_if_emptyBlock　　control warning

如果 if 语句没有 else 分枝，且其分枝内容为空，这样的 if 语句无任何意义，即使其条件有副作用，也不应继续保留该 if 结构。

示例：

```
if (a = foo());   // Non-compliant
if (a == bar()) {}   // Non-compliant
```

参考

CWE-1071

R9.1.13 if...else-if 分枝数量应在规定范围之内

ID_if_tooManyElseIf　　control warning

---

if...else-if 分枝超过指定数量，代码较为复杂不利于维护，而且在执行时各分枝的条件需逐一判断，效率较低，建议改为遵循某种算法的索引结构。

示例：

```
if (rabbit) {
}
else if (hamster) {
}
// ... 3000 branches...
// Computers have the courage to execute,
// but do you have the courage to read?
else {
}
```

建议 if...else-if 分枝数量不超过 5 个。

配置

maxElseIfCount：分枝数量上限，超过则报出

R9.1.14 if 分枝中的语句应该用大括号括起来

ID_if_brace　　control suggestion

组成 if 分枝的语句应为大括号括起来的复合语句，即使该复合语句只包含一条语句。

这是非常好的编程习惯，如果能一直遵循此规则，可以杜绝由错误缩进、宏的错误使用造成的多种问题。

示例：

```
if (cond1)         // Non-compliant
    if (cond2)     // Non-compliant
        action1();
else               // Non-compliant
    action2();
```

这段代码想表达的逻辑应是：

```
if (cond1) {
    if (cond2) {
        action1();
    }
} else {
    action2();
}
```

但 C/C++ 规定 else 子句与最近的 if 子句配对，所以实际逻辑是：

```
if (cond1) {
    if (cond2) {
        action1();
    } else {
        action2();
    }
}
```

这显然是与预期不符的。

又如：

```
if (cond)    // Non-compliant
    y = 2;
else         // Non-compliant
    x = 3;
    y = 1;
```

注意 else 子句中的 y = 1；由于缩进的关系此句看起来应该是 else 分枝的一部分，但它实际上并不在 else 的作用范围之内，所以用大括号括起来，可杜绝此类问题：

```
if (cond) {    // Compliant
    y = 2;
} else {       // Compliant
    x = 3;
    y = 1;
}
```

相关

ID_do_brace
ID_for_brace
ID_while_brace
ID_switch_brace

参考

MISRA C 2004 14.9
MISRA C 2012 15.6
MISRA C++ 2008 6-4-1

R9.1.15 所有 if...else-if 分枝都应以 else 子句结束

ID_if_missingEndingElse control suggestion

所有 if...else-if 分枝都以 else 子句结束是非常好的编程习惯，这与要求 switch 语句包含 defualt 分枝一样，是"防御性编程"思想的良好体现，参见 ID_switch_missingDefault。

单独的一个 if 分枝不要求接有 else 子句：

```
if (x > 0) {
    ....
}
```

存在多个 if...else-if 分枝时，要求接有 else 子句：

```
if (x > 0) {
    ....
}
else if (y < 0){
    ....
}
else {
    // Comment is the minimum requirement,
    // if here is unreachable logically,
    // it's better to log or throw an exception
}
```

相关

ID_switch_missingDefault

参考

MISRA C 2012 15.7
MISRA C++ 2008 6-4-2

9.2 For

R9.2.1 for 语句不应被分号隔断

ID_for_semicolon　　control error

---

for 语句不应被分号隔断。

示例：

```
for (....);  // Non-compliant, see the semicolon
{
    ....
}
```

分号使循环失效，这可能是笔误，也可能是有人恶意篡改了代码，应立即修正。

相关

ID_do_brace
ID_if_brace
ID_switch_brace
ID_while_brace

参考

CWE-670

R9.2.2 for 循环中不应存在无条件的跳转语句

ID_for_uncondBroken　　control error

无条件的 return、throw 或 break 语句会使循环失效，无条件的 continue 语句会使其后面的代码失效，如果其后没有代码时，该 continue 语句是没有意义的。

示例：

```
for (....) {
    if (cond)
        foo();
        break;  // Non-compliant
}
```

这种问题多数由错误的缩进或混乱的逻辑造成。

相关

ID_while_uncondBroken

参考

CWE-670

R9.2.3 for 语句作用域的范围不应有误

ID_for_scope control warning

---

for 语句作用域的范围不应有误。

示例：

```
for (....)
    statement1; statement2;  // Non-compliant

for (....)
    statement1;
    statement2;  // Non-compliant
```

例中 statement2 不在 for 循环的作用域中，但看起来又和 for 循环相关，这种问题多数是由宏展开或无效的缩进造成的。
为了避免这种问题，for 语句应使用大括号括起来。

相关

ID_for_brace

参考

CWE-483

R9.2.4 如果 for 语句没有明显的循环变量则应改为 while 循环

ID_for_simplification control suggestion

如果 for 迭代声明中的第 1 个和第 3 个表达式为空，应改为 while 循环，使代码更简洁。

示例：

```
for (;condition;)   // Non-compliant
```

不妨化简为：

```
while (condition)   // Compliant
```

例外：
for (;;) 被当作一种无限循环的惯用方法可被排除。

参考

C++ Core Guidelines ES.73

R9.2.5 for 循环体不应为空

ID_for_emptyBlock control warning

空的 for 循环将逻辑功能全部压缩到了迭代表达式中，可读性较差。

示例：

```
void foo(int n, vector<int>& v) {
    int i = 0;
    for (; i < n; i++);  // Non-compliant
    bar(i);              // The indent is odd here
    for (auto x: v);     // Non-compliant, meaningless
}
```

应改为：

```
void foo(int n, vector<int>& v) {
    int i = 0;
    for (; i < n; ) {     // Compliant, but a while-loop is better
        i++;
    }
    bar(i);
    for (auto x: v) {     // Compliant
        ....
    }
}
```

参考

C++ Core Guidelines ES.85

R9.2.6 for 循环变量不应为浮点型

ID_for_floatCounter　　control warning

用于控制循环次数的变量称为循环变量，这种变量不应采用浮点类型，否则循环的次数难以控制。

由于浮点型变量的不精确性使浮点型变量不适用于控制循环次数，相关讨论可参见 ID_illFloatComparison。

示例：

```
size_t n = 0;
for (float f = 0.f; f < 1.f; f += 0.001f) {  // Non-compliant
    ....
    n++;
}
cout << n << '\n';  // Not 1000
```

本例按常识应循环 1000 次，然而由于 f 无法精确表示 0.001，导致实际循环次数与预期产生偏差。

可变通地建立整形循环变量与浮点数的关系：

```
for (size_t n = 0; n < 1000; n++) {  // Compliant
    float f = n * 0.001f;
    ....
}
```

相关

ID_illFloatComparison

参考

MISRA C 2004 13.4
MISRA C 2012 14.1
MISRA C++ 2008 6-5-1

R9.2.7 for 循环变量不应在循环体内被改变

ID_for_counterChangedInBody control warning

用于控制循环次数的变量称为循环变量，这种变量只应在 for 迭代声明的第 3 个表达式中被改变，否则陡增逻辑复杂度，且可读性较差。

示例（选自 C++ Core Guidelines）：

```
for (int i = 0; i < 10; ++i) {
  // no updates to i -- ok
}
for (int i = 0; i < 10; ++i) {
  //
  if (/*something*/) ++i; // BAD
  //
}
bool skip = false;
for (int i = 0; i < 10; ++i) {
  if (skip) { skip = false; continue; }
  //
  if (/*something*/) skip = true; // Better:using two variables for two
concepts.
  //
}
```

参考

C++ Core Guidelines ES.86
MISRA C 2004 13.6
MISRA C++ 2008 6-5-3

R9.2.8 嵌套的 for 循环不应使用相同的循环变量

ID_for_counterNested　　control warning

同一个循环变量在内外层 for 循环中均被修改，使循环次数难以控制，是过于复杂的循环逻辑，也可能是某种错误。

示例：

```
for (int i = 0; i < 100; i++) {
    for (; i < 10; i++) {  // Non-compliant
        ....
    }
}
```

相关

ID_for_counterChangedInBody

R9.2.9 for 循环体应该用大括号括起来

ID_for_brace　　control suggestion

---

for 循环体应为复合语句，即使只包含一条语句。

示例：

```
int foo() {
    int a = 0;
    for (int i = 0; i < 10; i++)  // Non-compliant
        a += i;
    return a;
}
```

应改为：

```
int foo() {
    int a = 0;
    for (int i = 0; i < 10; i++) {  // Compliant
        a += i;
    }
    return a;
}
```

参考

MISRA C 2004 14.8
MISRA C 2012 15.6
MISRA C++ 2008 6-3-1

9.3 While

R9.3.1 while 语句不应被分号隔断

ID_while_semicolon　　control error

---

while 语句不应被分号隔断。

示例：

```
while (condition);  // Non-compliant, see the semicolon
{
    ....
}
```

分号使循环失效，有造成死循环的危险。

参考

CWE-670

R9.3.2 while 语句中不应存在无条件的跳转语句

ID_while_uncondBroken　　control error

不受条件限制的 return、throw 或 break 语句会使循环失效，不受条件限制的 continue 语句会使其后面的代码失效，如果其后没有代码，该 continue 语句是没有意义的。

示例：

```
while (condition) {
    ....
    return;     // Non-compliant
}

while (condition) {
    ....
    break;      // Non-compliant, becomes an if-statement
}
```

```
while (condition) {
    ....
    continue;   // Non-compliant, meaningless continue
}
```

这种问题多数由错误的缩进或混乱的逻辑造成。

相关

ID_for_uncondBroken

参考

CWE-670

R9.3.3 while 语句的条件不应为赋值表达式

ID_while_assignment　　control warning

虽然语言允许在 while 语句的条件中赋值，但 = 和 == 极易混淆，建议所有产生 bool 型结果的表达式，都不要包含赋值运算符。

示例：

```
while (x = 123)  // Non-compliant
{
    ....
}
```

相关

ID_if_assignment

参考

CWE-480
CWE-783

R9.3.4 while 语句作用域的范围不应有误

ID_while_scope control warning

---

while 语句作用域的范围不应有误。

示例：

```
while (condition)
    statement1; statement2;  // Non-compliant

while (condition)
    statement1;
    statement2;  // Non-compliant
```

例中 statement2 不在 while 循环的作用域中，但看起来又和 while 循环相关，这种问题多数是由宏展开或无效的缩进造成的。
为了避免这种问题，while 语句应使用大括号括起来。

相关

ID_while_brace

参考

CWE-483

R9.3.5 while 循环体不应为空

ID_while_emptyBlock control suggestion

空的 while 循环将逻辑功能全部压缩到了条件表达式中，可读性较差。

示例：

```
void foo(char* d, const char* s) {
    while (*d++ = *s++);  // Non-compliant
}
```

应将条件和运算分开，提高可读性：

```
void foo(char* d, const char* s) {
    int i = 0;
    while (s[i] != '\0') {  // Compliant
        d[i] = s[i];
        i += 1;
    }
    d[i] = '\0';
}
```

经过现代编译器的优化，这两种方式在效率上并无区别，而后者可读性更高。

参考

CWE-1071
C++ Core Guidelines ES.85

R9.3.6 while 循环体应该用大括号括起来

ID_while_brace　　control suggestion

---

while 循环体应为复合语句，即使只包含一条语句。

示例：

```
while (condition)     // Non-compliant
    statement;

while (condition)     // Non-compliant
    statement1;
    statement2;

while (condition) {   // Compliant
    statement;
}

while (condition) {   // Compliant
    statement1;
    statement2;
}
```

相关

ID_do_brace
ID_for_brace
ID_if_brace
ID_switch_brace
ID_switch_onlyDefault
ID_switch_onlyOneCase

参考

MISRA C 2004 14.8
MISRA C 2012 15.6
MISRA C++ 2008 6-3-1

9.4 Do

R9.4.1 注意 do-while(false) 中可疑的 continue 语句

ID_do_suspiciousContinue control warning

---

continue 语句和 break 语句在语义上是不同的，但在 do-while(false) 中它们的功效是一样的。

在 do-while(false) 的循环体中如果既有 break 语句又有 continue 语句，那么 continue 语句被误用的可能性较大。

示例：

```
int foo() {
    do {
        ....
        if (cond1) {
            break;
        }
        ....
        if (cond2) {
            continue;  // Rather suspicious
        }
        ....
    } while (false);
}
```

为了减少误解，建议在 do-while(false) 中只使用 break 语句，不使用 continue 语句。

参考

CWE-670

R9.4.2 do-while 循环体不应为空

ID_do_emptyBlock　　control suggestion

空的 do-while 循环将逻辑功能全部压缩到了条件表达式中，可读性较差。

示例：

```
void foo(char* d, const char* s) {
    do {} while (*d++ = *s++);      // Non-compliant
}
```

相关

ID_while_emptyBlock

参考

CWE-1071
C++ Core Guidelines ES.85

R9.4.3 do-while 循环体应该用大括号括起来

ID_do_brace　　control suggestion

---

do-while 循环体应为复合语句，即使只包含一条语句。如果没有合理的大括号，可能会与内嵌的 while 语句形成难以发觉的错误。

示例：

```
do    // Non-compliant
    foo();
    while (cond1);
while (cond2);
```

例中 while 关键字与 do 关键字产生了错误的对应关系，导致最后一个 while 形成了死循环，应改为：

```
do {  // Compliant
    foo();
    while (cond1) {
        ....
    }
} while (cond2);
```

相关

ID_for_brace
ID_if_brace
ID_switch_brace
ID_while_brace

参考

MISRA C 2004 14.9
MISRA C 2012 15.6
MISRA C++ 2008 6-4-1

R9.4.4 不建议使用 do 语句

ID_do_deprecated　　control suggestion

---

do 语句的终止条件在末尾，且第一次执行时不检查终止条件，可读性较低，不利于维护。

示例：

```
int foo(int n) {
    do {
        if (n < 0) {
            break;
        }
        ....
        if (n > 0) {
            continue;
        }
        ....
    } while (cond);  // Too complex
    ....
    return n;
}
```

do 语句糅合循环和流程跳转，使代码过于复杂，建议将复杂的 do 语句抽取成函数，使代码的结构更明确。

参考

C++ Core Guidelines ES.75

9.5 Switch

R9.5.1 switch 语句不应被分号隔断

ID_switch_semicolon　　control error

---

switch 语句不应被分号隔断。

示例：

```
switch (v);  // Non-compliant
```

这是毫无意义的 switch 语句，可能是残留代码，应及时去除。

参考

CWE-670

R9.5.2 switch 语句不应为空

ID_switch_emptyBlock　　control warning

空的 switch 语句没有意义。

示例：

```
switch (v) {}  // Non-compliant
```

这是毫无意义的 switch 语句，可能是残留代码，也可能是功能未实现。

参考

CWE-1071

R9.5.3 case 常量的范围不可超出 switch 变量的范围

ID_switch_caseOutOfRange control warning

如果 case 常量的范围超出了 switch 变量的范围，会导致相应分枝永远不会被执行。

示例：

```
void foo(char c)
{
    switch (c)
    {
    case 100:  // Compliant
        ....
        break;
    case 256:  // Non-compliant, unreachable branch
        ....
        break;
    }
}
```

例中变量 c 的值不可能为 256，所以 case 256 对应的分枝永远不会被执行。

相关

ID_illComparison

参考

CWE-561

R9.5.4 switch 语句中任何子句都应从属于某个 case 或 default 分枝

ID_switch_invalidStatement control error

---

switch 语句中任何子句都应从属于某个 case 或 default 分枝，否则不会被执行。

示例：

```
switch (v)
{
    int i;    // Non-compliant
    i = 0;    // Non-compliant
case 1:
    ....
    break;
default:
    bar(i);   // Logic error, ‘i’ is not initialized
    break;
}
```

例中对变量 i 的声明和赋值不从属于任何 case 或 default 分枝，是无效语句。

参考

CWE-561

R9.5.5 case 和 default 标签应直接从属于 switch 语句

ID_switch_badFormedCase　　control suggestion

不直接从属于 switch 语句的 case 或 default 标签用于非结构性跳转，是公认的不良实现。

关于非结构性跳转的进一步讨论可参见 ID_forbidGoto。

示例：

```
switch (v)
{
case 1:         // Compliant
    if (cond)
    {
case 2:         // Non-compliant
        ....
    }
    else
    {
default:        // Non-compliant
        ....
    }
}
```

例中 case 1 直接从属于 switch 语句，而 case 2 和 default 直接从属于 if 语句，当 v 的值不是 1 时，会绕过 if 语句的条件判断，产生非结构性跳转，与 goto 语句的问题一样，很容易导致逻辑混乱且不利于维护。

虽然有些编程技巧会将 case 置于循环中，如“Duff's device”等，但当今主流的编程语言均已不再提倡非结构性跳转。

相关

ID_forbidGotoBlocks

参考

MISRA C 2004 15.1
MISRA C++ 2008 6-4-4

R9.5.6 不应存在紧邻 default 标签的空 case 标签

ID_switch_uselessFallThrough　　control warning

紧邻 default 标签的空 case 标签是没有意义的，应当去除。

示例：

```
switch (v)
{
case 0:     // Compliant
    ....
    break;

case 1:     // Non-compliant
default:
case 2:     // Non-compliant
    ....
    break;
}
```

应改为：

```
switch (v)
{
case 0:     // Compliant
    ....
    break;

default:    // Compliant
    ....
    break;
}
```

R9.5.7 不应存在内容完全相同的 case 分枝

ID_switch_identicalBranch control warning

内容完全相同的分枝应合并为一个分枝，也可能是由复制粘贴造成的错误。

示例：

```
switch (v)
{
case 1:
    branch1
    break;
case 2:
    branch2
    break;
case 3:
    branch1   // Non-compliant
    break;
}
```

例中 case 3 对应的分枝和 case 1 对应的分枝内容完全相同，应将其合并为一个分枝，或修正本应存在的差异。

例外：
如果分枝内容较少，为了使代码更清晰可以接受适当的重复，但如果分枝内容很多就不应重复了，不妨指定一个数量限制，当重复分枝的符号数量超过这个限制时算作违规，否则放过。

配置

branchTokenCountThreshold：重复分枝符号数量限制，不检查符号数量小于该值的分枝

相关

ID_if_identicalBlock
ID_if_identicalElseIfBlock

参考

C++ Core Guidelines ES.3

R9.5.8 switch 语句的条件变量或表达式不应为 bool 型

ID_switch_bool control warning

switch 语句的条件表达式为 bool 型时不应采用 switch 语句，应采用 if-else 语句。

示例：

```
void foo(bool b)
{
    switch (b)   // Non-compliant
    {
    case true:
        ....
        break;
    case false:
        ....
        break;
    }
}
```

应改为：

```
void foo(bool b)
{
    if (b)  // Compliant
    {
        ....
    }
    else
    {
        ....
    }
}
```

参考

MISRA C 2004 15.4
MISRA C 2012 16.7
MISRA C++ 2008 6-4-7

R9.5.9 switch 语句不应只包含 default 标签

ID_switch_onlyDefault control warning

只有 default 标签的 switch 语句是没有意义的，起不到分枝选择的作用，往往是残留代码或功能未实现。

示例：

```
switch (v)
{
default:    // Non-compliant
    ....
}
```

这种空的可以 fallthrough 到 default 标签的空 case 标签也是没有意义的：

```
switch (v)
{
case 1:     // Non-compliant
case 2:
default:
    ....
}
```

参考

MISRA C 2012 16.6

R9.5.10 switch 语句不应只包含一个 case 标签

ID_switch_onlyOneCase　　control warning

只有一个 case 标签的 switch 语句与 if 语句语义相同，但形式上更为复杂，应改为 if 语句。

示例：

```
switch (v)
{
case 123:    // Non-compliant
    ....
    break;
}
```

应改为：

```
if (v == 123)
{
    ....
}
```

参考

MISRA C 2012 16.6

R9.5.11 switch 语句分枝数量应在规定范围之内

ID_switch_tooManyCases　　control warning

---

switch 语句分枝过多会使代码过于庞大不利于维护，分枝很多时建议将每个 case 的执行逻辑抽取成函数，再按遵循某种算法的索引结构组织在一起。

示例：

```
switch (v)
{
case 1: .... break;
case 2: .... break;
    // ... Lots of cases ...
case 1000: .... break;  // Non-compliant
}
```

建议 case 数量不超过 10 个。

配置

maxCasesCount：分枝数量上限，超过则报出

R9.5.12 switch 语句应配有 default 分枝

ID_switch_missingDefault　　control suggestion

所有 switch 语句都配有 default 分枝是非常好的编程习惯，这与 if...else-if 分枝要求有 else 分枝一样，是“防御性编程”思想的良好体现，参见 ID_if_missingEndingElse。

示例：

```
switch (v)
{
case 0:
    ....
    break;
case 1:
    ....
    break;
default:
```

```
    // Comment is the minimum requirement,
    // if here is unreachable logically,
    // it's better to log or throw an exception
    break;
}
```

例外：
当 switch 变量为枚举类型，且 case 标签已对应所有枚举项时，不再要求有 default 分枝。

相关

ID_if_missingEndingElse

参考

CWE-478
MISRA C++ 2008 6-4-6

R9.5.13 switch 语句的每个非空分枝都应该用无条件的 break 语句终止

ID_switch_breakOmitted　　control warning

每个非空分枝都应该用无条件的 break 语句终止，break 语句的缺失或误用是导致错误的常见原因。

示例：

```
switch (a)
{
case 0:
    b = 1;
    break;  // Compliant
case 1:
    b = 2;  // Non-compliant, missing ‘break’
default:
    b = 3;
    break;  // Compliant
}
```

相连的 case 标签不受本规则约束：

```
switch (c)
{
case 0:     // Compliant
case 1:
    ....
    break;
}
```

少数情况下，如果确实不能有 break 语句，应添加注释说明情况，或在 C++ 语言中用 [[fallthrough]] 注明：

```
switch (v)
{
case 1:
    do_something();
    [[fallthrough]];  // Compliant, since C++17
default:
    do_something_default();
    break;
}
```

依据

ISO/IEC 14882:2017 10.6.5

参考

CWE-484
C++ Core Guidelines ES.78
MISRA C 2004 15.2
MISRA C 2012 16.3
MISRA C++ 2008 6-4-5

R9.5.14 switch 语句应该用大括号括起来

ID_switch_brace　　control suggestion

---

switch 语句应为包含多条语句的复合语句，且用大括号括起来，否则不应选用 switch 语句。

示例：

```
switch (v)  // Non-compliant
    case 0:
        foo(v);
```

应改为 if 语句：

```
if (v == 0) {  // Compliant
    foo(v);
}
```

相关

ID_if_brace
ID_switch_onlyDefault
ID_switch_onlyOneCase

参考

MISRA C 2004 14.8
MISRA C 2012 15.6
MISRA C++ 2008 6-3-1

R9.5.15 switch 语句不应嵌套

ID_switch_forbidNest ⛔ control suggestion

嵌套的 switch 语句使代码显得复杂，不利于维护。

示例：

```
switch (u)
{
case 1:
    ....
case 2:
    switch (v)  // Non-compliant
    {
    case 3:     // Confusing
        ....
    case 4:
        ....
    }
default:
    ....
}
```

不同 switch 的 case 交织在一起，较难看出哪个 case 对应哪个变量，尤其是代码行数较多时这种问题会更为明显，应尽量不使用嵌套，或将内嵌的 switch 语句抽取成一个函数，使代码更清晰。

9.6 Try

R9.6.1 不应存在空的 try 块

ID_try_emptyBlock control warning

空的 try 块是毫无意义的，有可能是残留代码或功能未实现。

示例：

```
try {
    // Empty block or some code commented out
}
catch (Exception& e) {
    // The whole statement is meaningless
}
```

如果是残留代码应及时删去，否则引入无意义的异常处理会影响代码优化。

参考

CWE-1071

R9.6.2 catch 块序列中 catch-all 块（ellipsis handler）应位于最后

ID_try_disorderedEllipsis control error

---

catch 块序列中 catch-all 块（ellipsis handler）应位于最后，否则其后的 catch 块将失去作用。

示例：

```
try {
    foo();
} catch (...) {  // Non-compliant
    ....
} catch (const E&) {
    ....
}
```

应改为：

```
try {
    foo();
} catch (const E&) {
    ....
} catch (...) {  // Compliant
    ....
}
```

依据

ISO/IEC 14882:2003 15.3(6)
ISO/IEC 14882:2011 15.3(5)
ISO/IEC 14882:2011 18.3(5)

参考

CWE-561
C++ Core Guidelines E.31
MISRA C++ 2008 15-3-7

R9.6.3 catch 块序列中针对派生类的应排在前面，针对基类的应排在后面

ID_try_disorderedHandlers　　control error

---

catch 块序列中针对派生类的应排在前面，针对基类的应排在后面，如果违反这个顺序，针对派生类的 catch 块将失去作用。

示例：

```
class B { .... };
class D: public B { .... };

try {
    foo();
} catch (const B&) {
    ....
} catch (const D&) {  // Non-compliant, unreachable
    ....
}
```

应改为：

```
try {
    foo();
} catch (const D&) {
    ....
} catch (const B&) {  // Compliant
    ....
}
```

依据

ISO/IEC 14882:2003 15.3
ISO/IEC 14882:2011 15.3
ISO/IEC 14882:2011 18.3

参考

CWE-561
C++ Core Guidelines E.31

R9.6.4 try 块不应嵌套

ID_try_forbidNest　　⛔ control suggestion

嵌套的 try-catch 使代码显得复杂，不利于维护。

示例：

```
try {
    ....
    try {       // Non-compliant
        ....
    } catch (A&) {
        ....
    }
} catch (B&) {
    ....
}
```

嵌套的 try-catch 较难看出哪个 try 块对应哪个 catch 块，当代码行数较多时这种问题会更为明显。

参考

C++ Core Guidelines E.17

9.7 Catch

R9.7.1 通过引用捕获异常

ID_catch_value　　control warning

如果按传值的方式捕获异常会造成不必要的复制开销，也可能产生“对象切片”问题；如果通过指针捕获异常，会增加不必要的内存管理开销，通过引用捕获异常才是合理的方式。

示例：

```
try {
    ....
} catch (Exception e) {   // Non-compliant
    ....
}
```

例中 Exception 是异常类，用传值的方式捕获异常是不符合要求的。

应改为：

```
try {
    ....
} catch (Exception& e) {   // Compliant
    ....
}
```

通过指针捕获异常也是不符合要求的，可参见 ID_throwPointer 中的示例与讨论。

相关

ID_catch_slicing
ID_throwPointer

参考

C++ Core Guidelines E.15
C++ Core Guidelines ES.63
MISRA C++ 2008 15-3-5

R9.7.2 捕获异常时不应产生对象切片问题

ID_catch_slicing control warning

如果按传值的方式捕获多态类的异常对象，会使对象的多态性失效，造成错误的异常处理。

本规则是 ID_catch_value 与 ID_objectSlicing 的特化。

示例：

```
class Exception {
public:
    Exception();
    virtual ~Exception();
    virtual const char* what() const { return nullptr; }
};

void foo() {
    try {
        // Objects derived from Exception may be thrown...
    }
    catch (Exception e) {  // Non-compliant
        log(e.what());
    }
}
```

如果例中 Exception 类是所有异常类的基类，不论哪个异常被捕获，what 函数只能返回 nullptr，丧失了多态性，造成了异常的错误处理。

相关

ID_catch_value
ID_objectSlicing

参考

C++ Core Guidelines C.145
C++ Core Guidelines ES.63

R9.7.3 捕获异常后不应直接重新抛出异常，需对异常进行有效处理

ID_catch_justRethrow control warning

捕获异常后将其直接重新抛出是没有意义的，还会造成不必要的开销。

示例：

```
void foo() {
    try {
        bar();
    }
    catch (...) {  // Non-compliant
        throw;
    }
}
```

例中的 catch 块是没有意义的，应将其去掉。

R9.7.4 不应存在空的 catch 块

ID_catch_emptyBlock　　control suggestion

空的 catch 块掩盖了异常，不利于问题的排查与纠正，应至少添加日志记录等操作。

示例：

```
void foo() {
    try {
        ....
    }
    catch (...)
    {}  // Non-compliant, very bad
}
```

这样做并不能真正提高程序的稳定性，相当于逃避了问题，而且掩盖没有被处理的异常也可能会影响到其他方面的正常运行。

对于要求不能抛出异常的接口，不妨按下例处理，记录意料之外的异常情况，以便问题的排查：

```
void foo() noexcept {
    try {
        ....
    }
    catch (...) {  // Compliant
        log_unexpected_and_exit(__FILE__, __LINE__, "some messages");
    }
}
```

参考

CWE-1069
CWE-1071
CWE-391

R9.7.5 不应捕获过于宽泛的异常

ID_catch_generic control warning

捕获过于宽泛的异常如 std::exception、std::logic_error、std::runtime_error 等，使异常处理失去针对性，无法做到具体问题具体处理，而且很可能将本不应处理的异常一并捕获，造成混乱。

相关讨论详见 ID_throwGenericException。

示例：

```
try {
    ....
} catch (std::exception&) {  // Non-compliant
    ....
}

try {
    ....
} catch (std::runtime_error&) {  // Non-compliant
    ....
}

try {
    ....
} catch (std::logic_error&) {  // Non-compliant
    ....
}
```

相关

ID_throwGenericException

参考

CWE-396

R9.7.6 不应捕获非异常类型

ID_catch_nonExceptionType control warning

字符串或变量以及非异常相关的对象不应被当作异常捕获，否则意味着异常相关的设计是不健全的。

相关讨论详见 ID_throwNonExceptionType。

示例：

```
try {
    ....
} catch (int) {  // Non-compliant
    ....
}

try {
    ....
} catch (char*) {  // Non-compliant
    ....
}

try {
    ....
} catch (string&) {  // Non-compliant
    ....
}
```

相关

ID_throwNonExceptionType

参考

C++ Core Guidelines E.14

10. Expression

10.1 Logic

R10.1.1 不应出现不合逻辑的重复子表达式

ID_illIdentical expression error

逻辑与、逻辑或、按位与、按位或的子表达式以及三元表达式的两个分枝不应重复，否则失去逻辑意义。

示例：

```
bool foo(int* p, char* s) {
    return p != NULL && p != NULL;  // Non-compliant
}

long bar() {
    return FLAG0 | FLAG1 | FLAG0;   // Non-compliant
}

char baz(bool cond) {
    return cond? 'a': 'a';          // Non-compliant
}
```

例中重复的子表达式都是有问题的，这是一种很常见的错误。

这种错误往往由复制粘贴引起，所以修正时不要只是删去重复的子表达式，要考虑是否有某些项被漏掉。

例外：
重复的子表达式有一定副作用时可不受本规则限。

```
if (fin.get() == 'a' && fin.get() == 'a') {  // Let it go
    ....
}
```

例中逻辑表达式从文件流中读取相邻的字符，但第二个子表达式可能不会被执行，这种代码即使没有逻辑错误也是不利于维护的，进一步讨论可参见 ID_shortCircuitSideEffect。

参考

CWE-682

R10.1.2 逻辑表达式中各子表达式不应自相矛盾

ID_conflictCondition　　expression error

在逻辑表达式中，相互矛盾的子表达式会使整个表达式的结果恒为真或恒为假，造成条件失效。

1. 判断同一变量同时等于不同的值是无效的：

```
a == 1 && a == 2  // always false
a != 1 || a != 2  // always true
```

2. 判断同一变量的上限小于下限是无效的：

```
a < -128 && a > 127    // always false
a >= -128 || a <= 127  // always true
```

3. 如下逻辑判断也是无效的：

```
a > b && a < b    // always false
a > b && a <= b   // always false
a > b || a <= b   // always true
a == b && a != b  // always false
a == b || a != b  // always true
```

这类问题均为常见笔误，须认真对待。

参考

CWE-570
CWE-571

R10.1.3 条件表达式不应恒为真或恒为假

ID_invalidCondition　　expression warning

变量初始化后不经修改即作为条件往往意味着某种逻辑错误。

示例：

```
void foo() {
    int i = 0;
    if (i > 0) {  // Non-compliant
        ....
    }
}
```

例中变量 i 初始化为 0 后在没有被修改过的情况下，仍对其进行判断是没有意义的，这显然是逻辑错误。

相关

ID_constLogicExpression

参考

CWE-570
CWE-571
MISRA C 2004 13.7
MISRA C 2012 14.3

R10.1.4 不应使用多余的逻辑子表达式

ID_redundantCondition　　expression warning

逻辑或、逻辑与的子表达式有包含关系时，其中的一个表达式是多余的。

示例：

```
a > b && a != b     // None-compliant, a != b is redundant
a < b || a != b     // None-compliant, a < b is redundant
a >= b && a == b    // None-compliant, a >= b is redundant
a == b || a <= b    // None-compliant, a == b is redundant
```

这种多余的子表达式很可能包含某种错误，需认真对待。

R10.1.5 逻辑表达式及其子表达式的结果不应为常量

ID_constLogicExpression　　expression warning

逻辑表达式及其子表达式的结果不应为常量，否则使逻辑判断失去意义。

示例：

```
if (false) {  // Non-compliant
    ....
}

while (false) {  // Non-compliant
    ....
}

const bool False = false;
if (False && other_condition) {  // Non-compliant
    ....
}
```

这种代码往往是调试或维护后的残留代码，没有意义的控制语句应及时去除。

例外：
true 或 1 等常量可用在 while 或 do-while 循环的条件中，false 或 0 等常量可用在 do-while 循环的条件中。

```
while (true) {  // Compliant
    ....
}

do {
    ....
} while (0);  // Compliant
```

但这种特例不包括逻辑表达式的子表达式，如：

```
while (other_condition || 1) {  // Non-compliant
    ....
}
```

另外，预编译阶段定义的常量也不应该作为逻辑条件或逻辑子表达式，如：

```
#define M 123

if (M) {  // Non-compliant
    ....
}
```

能在预编译阶段确定的代码应统一用预编译方式处理，不应占用运行时资源：

```
#if M  // Compliant
    ....
#endif
```

或：

```
if constexpr (M) {  // Compliant, since C++17
    ....
}
```

参考

CWE-570
CWE-571
MISRA C 2004 13.7
MISRA C 2012 14.3

R10.1.6 逻辑表达式的右子表达不应有副作用

ID_shortCircuitSideEffect　　expression suggestion

逻辑表达式的右子表达有副作用会使代码复杂度显著增加，易产生错误，且不利于维护。

对于逻辑表达式的求值，标准规定从左至右计算各子表达式的值，当可以确定整个表达式的值时，即使还有未计算的子表达式，也会立即结束求值，这种方法可提高效率，称为“短路规则（short-circuit evaluation）”。

逻辑表达式的右子表达式受左子表达式影响，可能不会被执行，如果有副作用也可能不会生效。

示例：

```
if (a == foo || b == bar++) {  // Non-compliant
    do_something(bar);         // Consider that ‘bar++’ may not be evaluated
}
```

如果 a == foo 为真，不论 b 是否等于 bar++，整个条件表达式的值一定为真，所以 b == bar++ 不一定会被执行，if 语句要同时考虑 bar++ 执行与未执行的两种状态，复杂度较高。

依据

ISO/IEC 9899:1999 6.5.13(4) 6.5.14(4)
ISO/IEC 9899:2011 6.5.13(4) 6.5.14(4)

参考

MISRA C 2004 12.4
MISRA C 2012 13.5
MISRA C++ 2008 5-14-1

R10.1.7 逻辑表达式应保持简洁明了

ID_simplifiableCondition　　expression warning

逻辑或、逻辑与的子表达式可以合并成一个表达式时应尽量合并。

示例：

```
a < b || a > b      // None-compliant, use a != b instead
a > b || a == b     // None-compliant, use a >= b instead
a < b || a == b     // None-compliant, use a <= b instead
a <= b && a >= b    // None-compliant, use a == b instead
a > b && a >= b     // None-compliant, use a >  b instead
a < b || a <= b     // None-compliant, use a <= b instead
```

这种不合常理的繁琐写法也可能包含某种错误，需认真对待。

R10.1.8 可化简为逻辑表达式的三元表达式应尽量化简

ID_simplifiableTernary　　expression suggestion

当三元表达式的分枝是 true 或 false 时可化简为逻辑表达式，应化简代码。

示例：

```
void foo(int a) {
    if (a > 123? true: false) {    // Non-compliant, too verbose
        ....
    }
}

bool bar(int a) {
    return a > 123? fun(): false;  // Non-compliant, verbose
}
```

应改为：

```
void foo(int a) {
    if (a > 123) {
        ....
    }
}

bool bar(int a) {
    return a > 123 && fun();
}
```

10.2 Evaluation

R10.2.1 避免依赖特定的求值顺序

ID_evaluationOrderReliance　　expression warning

不同的求值顺序不应产生不同的结果，否则极易导致意料之外的错误，也会降低代码的可移植性。

C 语言标准用“序列点（sequence point）”定义求值顺序，序列点前面的表达式先于后面的表达式求值并落实相关副作用，逻辑与、逻辑或、三元、逗号等运算符以及函数调用的左括号与序列点相关，其左子表达式先于右子表达式求值并落实副作用，赋值、算术、位运算符与序列点无关，其左右子表达式的求值顺序是未声明的，函数各参数的求值顺序也是未声明的，C++ 标准与 C 标准大致相同，C++17 明确了赋值、移位等运算符的求值顺序。

要注意子表达式的副作用在不同求值顺序下的正确性，可参见 ID_confusingAssignment 的进一步说明。

示例：

```
Stack s {1, 2, 3};      // A stack, the top is 1
int pop();              // Pop and return the top element from the stack
int x = pop() - pop();  // Non-compliant
```

设 pop 函数弹出并返回栈顶元素，减号左右的两个 pop 函数哪个先执行呢？这是标准未声明的，x 的值可以是 1 - 2，也可以是 2 - 1，由编译器决定。

应改为：

```
int a = pop();
int b = pop();
x = a - b;      // Compliant, or ‘b – a’, depends on your needs
```

这样便确定是栈项的第一个元素减第二个元素。

又如：

```
fun(pop(), pop());  // Non-compliant
```

设 fun 是函数名称或获取函数指针的表达式，标准规定 fun 会先于参数求值，但参数之间的求值顺序是未声明的。

逻辑与、逻辑或、三元、逗号等表达式可不受本规则限制，但其子表达式需受本规则限制。

相关

ID_confusingAssignment

依据

ISO/IEC 9899:2011 5.1.2.3(3)
ISO/IEC 9899:2011 Annex C

参考

CWE-758
C++ Core Guidelines ES.43
C++ Core Guidelines ES.44

R10.2.2 不应多次读写同一对象

ID_confusingAssignment　　expression warning

在表达式中多次引用并修改同一对象，很可能会因为非预期的求值顺序而产生错误的结果。

关于对象的副作用在求值过程中何时生效这一问题，相关标准既复杂又有大量未声明和未定义的情况，故需注意：

1. 写入对象的次数不应超过 1 次
2. 对象不应既被读取又被写入，除非是为了计算对象的新状态并写入对象

注意，对 volatile 对象的读取相当于更新对象的值，也是一种副作用，故：

1. volatile 对象在表达式中只应出现 1 次
2. volatile 对象不应既被读取又被写入

本规则是 ID_evaluationOrderReliance 的特化。

示例：

```
a = a++;        // Non-compliant
a = ++a;        // Non-compliant

++b = b;        // Non-compliant
b = a++ + a;    // Non-compliant

arr[i] = ++i;   // Non-compliant
p->fun(p++);    // Non-compliant

fun(a, a++);    // Non-compliant
fun(++a, a++);  // Non-compliant
```

例中 ++ 泛指写入操作。

设 a 是值为 0 的整型变量，如下表达式：

```
a = a++;        // Non-compliant
```

对变量 a 有两次写入，分别是增 1 和赋值为 0（子表达式 a++ 的值为 0），这两次写入的次序在 C 和 C++17 之前的标准中是未声明的，如果先增 1 再赋 0，a 的值最终为 0，如果先赋 0 再增 1，a 的值最终为 1，这种不确定的结果应当避免，C++17 规定了右子表达式的副作用先于赋值生效，所以在 C++17 之后例中表达式是无效的。

虽然新的标准强化了求值顺序，但这种代码使人费解，很容易造成理解上的偏差，故不应使用。

如果 a 不是 volatile 变量，应改为：

```
a++;            // Compliant, or
a += 1;         // Compliant, or
a = a + 1;      // Compliant
```

如果 a 是 volatile 变量，应改为：

```
int tmp = a;
a = tmp + 1;    // Compliant
```

对于逻辑与、逻辑或、三元以及逗号表达式，标准明确规定了子表达式从左至右求值，左子表达式的副作用也会在右子表达式求值前生效，故可不受本规则限制，但其子表达式仍需受本规则限制，进一步可参见“序列点（sequence point）”以及“求值顺序”等概念。

相关

ID_evaluationOrderReliance

依据

ISO/IEC 9899:2011 6.5(2)-undefined
ISO/IEC 14882:2003 5(4)-unspecified
ISO/IEC 14882:2011 1.9(15)-undefined
ISO/IEC 14882:2011 5.17(1)
ISO/IEC 14882:2017 8.18(1)
ISO/IEC 9899:2011 Annex C

参考

C++ Core Guidelines ES.43
SEI CERT EXP50-CPP
MISRA C++ 2008 5-0-1
MISRA C 2012 13.2

R10.2.3 注意运算符优先级，不可产生非预期的结果

ID_unexpectedPrecedence　　expression warning

对运算符优先级的错误理解是产生逻辑错误的主要原因之一。

示例：

```
int foo(bool cond) {
    return 1 + cond? 2: 3;  // Rather suspicious
}
```

加号的优先级大于三元运算符，但 cond 是 bool 型变量，所以这种情况十分可疑。

很可能应改为：

```
int foo(bool cond) {
    return 1 + (cond? 2: 3);
}
```

参考

CWE-783

R10.2.4 不在同一数组中的指针不可比较或相减

ID_illPtrDiff　　expression warning

不在同一数组中的指针比较或相减属于逻辑错误，会导致标准未定义的问题。

示例：

```
ptrdiff_t foo() {
    int a = 0, b = 0;
    int arr0[10] = {};
    int arr1[10] = {};

    if (&a < &b) {  // Non-compliant
        ....
    }
    return &arr1[5] - arr0;  // Non-compliant
}
```

依据

ISO/IEC 14882:2003 5.7(6)-undefined
ISO/IEC 14882:2011 5.7(6)-undefined
ISO/IEC 14882:2011 8.7(5)-undefined

参考

MISRA C 2004 17.3
MISRA C++ 2008 5-0-17
MISRA C++ 2008 5-0-18
C++ Core Guidelines ES.62

R10.2.5 bool 型变量或表达式不应参与大小比较、位运算、自增自减等运算

ID_illBoolOperation　　expression warning

bool 值只能为真或假，不具有“大小”这种逻辑意义，所以 bool 型变量或表达式参与大小比较、位运算、自增自减等运算都是不合理的。

示例：

```
bool foo(unsigned flags, unsigned flag) {
    return flags & flag != 0;  // Non-compliant
}
```

由于 != 的优先级高于 &，所以例中的 return 语句相当于先判断 flag 是否为 0，再将这个 bool 型的结果与 flags 按位与，这是没有意义的。

应改为：

```
bool foo(unsigned flags, unsigned flag) {
    return (flags & flag) != 0;  // Compliant
}
```

依据

ISO/IEC 14882:2011 5.3.2(1 2) D.1-deprecated

参考

CWE-1024
CWE-1025
CWE-682
CWE-783
MISRA C 2004 12.7
MISRA C 2012 10.1
MISRA C++ 2008 5-0-21

R10.2.6 不应出现复合赋值的错误形式

ID_illFormedCompoundAssignment　　expression warning

如下形式的复合赋值表达式（设 a 和 x 为变量或表达式）：

```
a -= a - x;
a /= a / x;
a &= a & x;
a |= a | x;
a ^= a ^ x;
```

是没有意义的，均为常见笔误，应将复合赋值改为普通赋值，或去掉重复的子表达式。

参考

CWE-682

R10.2.7 避免出现复合赋值的可疑形式

ID_suspiciousCompoundAssignment　　expression suspicious

如下形式的复合赋值表达式（设 a 和 x 为变量或表达式）：

```
a += a + x;
a *= a * x;
a %= a % x;
a <<= a << x;
a >>= a >> x;
```

均为常见笔误，但在特定需求下也有其逻辑意义，故对这种表达式应给出可疑提醒。即使这类表达式没有逻辑错误，也应该换成普通赋值表达式，以提高可读性。

示例：

```
a += a + x;  // Rather suspicious
```

应改为：

```
a = a + x;  // OK
a = 2 * a + x;  // OK
a = a + (a + x);  // OK
```

参考

CWE-682

R10.2.8 &=、|=、-=、/=、%= 左右子表达式不应相同

ID_illSelfCompoundAssignment　　expression warning

&=、|= 左右子表达式如果相同则没有任何效果，-=、/=、%= 左右子表达式相同则结果总为 1 或 0，这种表达式往往意味着笔误或逻辑错误。

示例（设 a 为变量或表达式）：

```
a &= a;  // Non-compliant, no effect
a |= a;  // Non-compliant, no effect
```

如果目的是清零或置 1，也不建议使用下列表达式：

```
a -= a;  // Non-compliant, tedious
a /= a;  // Non-compliant, low efficiency
a %= a;  // Non-compliant, low efficiency
```

对于高级语言来说，应该直接将变量赋值为 0 或 1，而不是采用更繁琐甚至低效的方式。

参考

CWE-682

R10.2.9 不应使用 NULL 对非指针变量赋值或初始化

ID_oddNullAssignment　　expression warning

标识符 NULL 只应该用来表示空指针，否则会对代码阅读造成误导，而且也可能是书写错误。

示例：

```
const int i = NULL;         // Non-compliant
const char c = NULL;        // Non-compliant
void foo(bool x = NULL);    // Non-compliant
```

应改为：

```
const int i = 0;            // Compliant
const char c = '\0';        // Compliant
void foo(bool x = false);   // Compliant
```

参考

MISRA C++ 2008 4-10-1

R10.2.10 赋值运算符与一元运算符之间应有空格，一元运算符与变量或表达式之间不应有空格

ID_stickyAssignmentOperator　　expression warning

如果 = 与 +、-、*、!、&、~ 等一元运算符之间没有空格，而一元运算符与其子表达式之间有空格，是一种非常怪异的格式，也可能是 +=、-=、*=、&=、~= 等复合赋值运算符的笔误。

示例：

```
a =+ b;    // Non-compliant
a =- b;    // Non-compliant
a =~ b;    // Non-compliant
a =! b;    // Non-compliant

a += b;    // Compliant
a = -b;    // Compliant
a ~= b;    // Compliant
a = !b;    // Compliant
```

参考

CWE-480

R10.2.11 赋值运算符左右子表达式不应重复

ID_selfAssignment　　expression warning

对自身赋值是没有逻辑意义的，往往是笔误或残留代码。

示例：

```
a = .... = b = a = ....  // Non-compliant
```

也可能是对语言特性不了解所致，如：

```
class A {
    int a;

public:
    A(int a) {
        a = a;   // Non-compliant, ‘a’ is not the member
    }
};
```

例中构造函数对成员 a 的赋值是无效的，应改为 this->a = a;

有时这种代码是为了消除编译警告，编译器可能会报出没有被用到的函数参数，有人将参数赋值给自身从而去除警告，但毕竟引入了没有实际意义的代码，改进方法可参见 ID_paramNotUsed。

有时为了设置调试断点，但又找不到合适的位置，于是增加了这种代码作为断点，显然这种非正式的代码是不应被保留的。

参考

CWE-682

R10.2.12 除法运算符、求余运算符左右子表达不应重复

ID_selfDivision expression warning

除法运算符、求余运算符左右子表达重复，结果总为 1 或 0 以及产生除零异常，这是没有意义的，往往是某种笔误。

示例：

```
int foo(int* p) {
    return p[0] % p[0];  // Non-compliant
}
```

参考

CWE-682

R10.2.13 减法运算符左右子表达式不应重复

ID_selfSubtraction expression warning

与自身做减法，结果总为 0，往往是某种笔误。

示例：

```
ptrdiff_t distance(const int* p0, const int* p1) {
    return p0 - p0;  // Non-compliant
}
```

例中减法表达式是没有意义的，很可能是 p1 被误写成了 p0，应改为：

```
ptrdiff_t distance(const int* p0, const int* p1) {
    return p0 - p1;  // Compliant
}
```

参考

CWE-682

R10.2.14 异或运算符左右子表达式不应重复

ID_selfExclusiveOr　　expression warning

与自身异或，结果总为 0，而且可能意味着某种错误。

对变量的清零，有一种惯用写法：

```
a = a ^ a;   // Non-compliant
```

这种复杂的写法在 C/C++ 等高级语言中已不再提倡，a ^= a 也不再提倡，应将变量直接赋值为 0，编译器会作更好的优化。

参考

CWE-682

R10.2.15 负号不应作用于无符号整数

ID_minusOnUnsigned　　expression warning

负号作用于无符号整数，结果仍是无符号整数，令人费解易产生意料之外的错误。

示例：

```
unsigned int a = 1;
long long b = -a;    // Non-compliant, b is 4294967295, confusing
```

例外：
unsigned char、unsigned short 等可以“类型提升”为 int 的无符号类型可被放过。
-1U、-1UL、-1ULL 作为 UINT_MAX、ULONG_MAX、ULLONG_MAX 的惯用简写形式可被放过。

依据

ISO/IEC 9899:1999 6.5.3.3(3)
ISO/IEC 9899:2011 6.5.3.3(3)

参考

MISRA C 2004 12.9
MISRA C 2012 10.1
MISRA C++ 2008 5-3-2

R10.2.16 不应重复使用一元运算符

ID_repeatedUnaryOperators　　expression warning

重复的一元运算符没有意义，为常见笔误。

示例：

```
int a = 1;
int b = ~~a;    // Non-compliant
int c = -+a;    // Non-compliant
int d = - -a;   // Non-compliant
```

例外：
两个连续的 ! 运算符是向 bool 型转换的惯用方法，不算违规，如：

```
bool e = !!a;   // Compliant
```

但如果超过两个 ! 还是会算作违规，如：

```
bool f = !!!a;  // Non-compliant
```

R10.2.17 运算结果不应溢出

ID_evalOverflow　　expression warning

运算结果超出对应类型的存储范围往往意味着错误。

这种情况对于有符号整数，会引发标准未定义的行为，而对于无符号整数，则只保留有效范围内的值，相当于一种模运算，标准认为这是一种“语言特性”，规定无符号整数不存在溢出，然而实践表明，运算结果超出无符号整数的范围很容易引起意料之外的问题，所以不论是否有符号，均应规避这种问题。

示例：

```
uint64_t a = 0xffffffffU + 1;  // Non-compliant
```

例中 0xffffffffU 是 32 位无符号整数的最大值，根据 C/C++ 语言的计算规则，0xffffffffU + 1 仍是 32 位无符号整数，不会自动转为 64 位整数，所以 a 的值是 0，而不会是 0x100000000。

应改为：

```
uint64_t a = 0xffffffffULL + 1;  // Compliant
```

又如：

```
int32_t a = foo();
int32_t b = bar();
int64_t c = a * b;  // Rather suspicious
```

例中 a 和 b 是 32 位整数，a*b 仍为 32 位整数，如果 a*b 的预期结果超过了 32 位就会造成溢出，这也是很常见的错误。

应改为：

```
int64_t c = static_cast<int64_t>(a) * b;  // OK
```

依据

ISO/IEC 9899:2011 6.5(5)-undefined
ISO/IEC 9899:2011 6.2.5(9)

参考

CWE-190
C++ Core Guidelines ES.103
C++ Core Guidelines ES.104

R10.2.18 位运算符不应作用于有符号整数

ID_bitwiseOperOnSigned　　expression warning

符号位在位运算方面没有逻辑意义，对负数进行位运算往往意味着逻辑错误。

示例：

```
int foo(signed s, unsigned u) {
    return s & u;  // Non-compliant
}

int bar(signed s, unsigned u) {
    if (s < 0) {
        int a = s << u;  // Non-compliant, undefined
        int b = s >> u;  // Non-compliant, implementation-defined
        return a + b;
    }
    return 0;
}
```

例中变量 s 为有符号类型，对其符号位的位运算是没有意义的，而且要注意对负数左移会导致未定义的行为，对负数右移则由实现定义。

依据

ISO/IEC 14882:2011 5.8(2)-undefined
ISO/IEC 14882:2011 5.8(3)-implementation

参考

CWE-682
MISRA C 2004 12.7
MISRA C 2012 10.1
MISRA C++ 2008 5-0-21
C++ Core Guidelines ES.101

R10.2.19 移位数量不可超过相关类型比特位的数量

ID_illShiftCount　　expression error

移位数量不可过大，否则会导致标准未定义的错误。

示例：

```
uint64_t foo(uint32_t u) {
    return u << 32;  // Non-compliant
}
```

例中 u 为 32 位整型变量，将其左移 32 位并不能得到 64 位的整数，其结果一般为 0，取决于编译器的具体实现。

应改为：

```
uint64_t foo(uint32_t u) {
    return static_cast<uint64_t>(u) << 32;  // Compliant
}
```

依据

ISO/IEC 14882:2003 5.8(1)-undefined
ISO/IEC 14882:2011 5.8(1)-undefined
ISO/IEC 14882:2017 8.8(1)-undefined

参考

MISRA C++ 2008 5-8-1

R10.2.20 逗号表达式的子表达式应具有必要的副作用

ID_invalidCommaSubExpression　　expression warning

逗号表达式的子表达式应具有必要的副作用，否则没有意义。

示例：

```
void foo(int& a, int& b) {
    a, b = 0, 1;   // Non-compliant
}
```

例中逗号表达式有 3 个子表达式，只有第 2 个子表达式有效，第 1 和第 3 个没有意义。

应改为：

```
void foo(int& a, int& b) {
    a = 0, b = 1;  // Compliant, but bad
}
```

本规则不建议使用逗号表达式，将逗号表达式拆分成合理的语句是更好的选择。

```
void foo(int& a, int& b) {
    a = 0;
    b = 1;         // Compliant, good
}
```

相关

ID_forbidCommaExpression

10.3 Comparison

R10.3.1 比较运算应在正确的范围内进行

ID_illComparison　　expression error

应在正确的范围内进行比较，否则会造成恒为真或恒为假的无效结果。

示例：

```
void foo(string& txt, string& sub) {
    size_t n = txt.find(sub);
    if (n >= 0) {  // Non-compliant, always true
        ....
    }
}
```

无符号变量不可能小于 0，也一定大于等于 0，例中 n >= 0 恒为真，是没有意义的条件。

又如：

```
typedef unsigned short X;
void fun(X x) {
    if (x == -1) {  // Non-compliant, always false
        ....
    }
}
```

例中 x 为无符号短整型变量，其取值范围为 [0, 65535]，x == -1 恒为假。由于“类型提升”，x 会被转为 int 型再与 -1 比较，x 恒为正数，-1 为负数，故不可能相等。

又如，对于有符号字符型变量，与其比较的数值不在 [-128, 127] 范围内时，也是无效的：

```
CodePage encodingDetect(const char* src) {
    char b0 = src[0];
    char b1 = src[1];
    char b2 = src[2];
    if (b0 == 0xef && b1 == 0xbb && b2 == 0xbf) {  // Non-compliant, always
false
        return cpUtf8;
    }
    ....
}
```

即使例中 b0 的二进制绝对值确实为 0xef，但由于“类型提升”，b0 转为 int 型后为负数，0xef 为正数，比较的结果恒为假。char 型变量是否有符号由实现定义，可参见 ID_plainNumericChar 的进一步说明，将 b0 等变量设为 unsigned char 可解决这个问题。

相关

ID_switch_caseOutOfRange

参考

CWE-697
CWE-1024
CWE-1025

R10.3.2 不应使用 == 或 != 判断浮点数是否相等

ID_illFloatComparison　　expression warning

一般来说，除了可以记作 a * 2^n（a、n 为整数）的浮点数值可以被精确存储之外，其他均为近似值。用 == 或 != 判断浮点数（float、double、long double）是否相等往往得不到预期的结果。

如 0、1、2、3、1.5、1.25、...可以被精确存储，而除此之外绝大部分数值如 0.1、0.2、0.3、...只能存储其近似值。

示例：

```
float f = 1;
f /= 10;
if (f == 0.1) {   // Non-compliant, do not use ‘==’ or ‘!=’
    cout << "OK";
} else {
    cout << "Oops";
}
```

输出 Oops，这是因为 f == 0.1 在运算时将变量和常量转为 double 型，而转换过程中生成了两个不同的对 0.1 的近似值（如 0.10000000149011611938 和 0.1000000000000000555），其结果自然为 false。
这非常容易造成意料之外的混乱，所以判断浮点数是否相等不应直接使用 == 或 != 运算符，即使浮点数可以被精确存储。

解决方法：

```
bool feq(float a, float b, float e = 0.0001f) {
    return fabs(a - b) < e;
}
```

利用 feq 函数判断浮点数是否相等，如果两个浮点数的差值非常小则可以认为相等，其中 fabs 为计算浮点数差值绝对值的函数，如果差值绝对值小于 e 则认为相等，否则不等。

```
if (feq(f, 0.1)) {  // Compliant
    cout << "OK";
}
```

参考

CWE-1025
MISRA C 2004 13.3
MISRA C++ 2008 6-2-2

R10.3.3 指针不应与字符串常量直接比较

ID_illPtrStrComparison　　expression error

直接比较指针和字符串常量的结果往往总是 false，应改用字符串比较函数。

示例：

```
bool is_name(const char* p) {
    return p == "bar";  // Non-compliant
}
```

如果例中 is_name 函数只接受常量字符串作为参数，该函数在某些环境中也可能正常工作，如：

```
if (is_name("foo")) {   // May be false
    ....
}
if (is_name("bar")) {   // May be true
    ....
}
```

然而相同的字符串常量是否一定拥有相同的地址呢？对这个问题不同的编译器有不同的实现，有可移植性要求的代码要规避这种问题，而且这种问题极易导致错误，一般的程序都应该避免这种问题。

应改为：

```
bool is_name(const char* p) {
    return !strcmp(p, "bar");  // Compliant
}
```

依据

ISO/IEC 14882:2003 2.13.4(2)-implementation
ISO/IEC 14882:2011 2.14.5(12)-implementation
ISO/IEC 14882:2011 5.13.5(16)-unspecified

参考

CWE-595
CWE-697
CWE-1024
CWE-1025

R10.3.4 不应比较非同类枚举值

ID_differentEnumComparison　　expression warning

比较非同类枚举值相当于比较不同类别的事物，没有逻辑意义，往往是设计缺陷或逻辑错误。

示例：

```
enum Fruit { apple, orange, banana };
enum Pet { hamster, chipmunk, chinchilla };

void foo(Fruit f) {
    if (f == hamster) {  // Non-compliant
        ....
    }
}

void bar(Pet p) {
    switch (p) {
    case apple:     // Non-compliant
        ....
        break;
    case chipmunk:  // Compliant
        ....
        break;
    }
}
```

R10.3.5 比较运算符左右子表达式不应重复

ID_selfComparison　　expression warning

与自身的比较是没意义的，往往是某种笔误。

示例（设 a 为变量或表达式）：

```
a == a   // Non-compliant
a != a   // Non-compliant
a > a    // Non-compliant
a >= a   // Non-compliant
a < a    // Non-compliant
a <= a   // Non-compliant
```

例外：
如果 a 为浮点数类型，判断 a 是否为无效值“NaN”的惯用方法是判断 a != a 是否为真，对于这种情况可以放过。

参考

CWE-1025

R10.3.6 比较运算不可作为另一个比较运算的直接子表达式

ID_successiveComparison　　expression error

在 C/C++ 语言中，连续的比较运算是没有意义的，本规则是 ID_illBoolOperation 的特化。

示例：

```
bool foo(int a, int b, int c) {
    return a >= b >= c;  // Non-compliant, logic error
}
```

例中 a >= b 的结果为 bool 型，bool 型数据是没有大小概念的，这个结果再与 c 比较大小是没有意义的。

如果是判断两个 bool 表达式是否相等，可以被本规则放过，如：

```
a > b == true       // Compliant, but odd
a != b != true      // Compliant, but odd
(a > b) == (c > d)  // Compliant, but complex
```

这种表达式虽然没有逻辑错误，但其复杂的形式并不值得提倡。

相关

ID_illBoolOperation

参考

CWE-697
CWE-1024
CWE-1025

10.4 Call

R10.4.1 返回值不应被忽略

ID_returnValueIgnored　　expression warning

返回值不应被忽略，尤其是与资源分配、信息获取、状态判断有关的返回值。

示例：

```
void foo(string& s) {
    s.empty();     // Non-compliant
}

void bar(unique_ptr<int[]>& p) {
    p.release();   // Non-compliant
}
```

另外，C++ 中由用户添加的具有 [[nodiscard]] 属性的函数，返回值也不应被忽略，如：

```
[[nodiscard]] int getStatus();

void baz() {
    getStatus();   // Non-compliant
}
```

参考

MISRA C 2012 17.7
MISRA C++ 2008 0-1-7

R10.4.2 不可臆断返回值的意义

ID_wrongUseOfReturnValue　　expression error

对接口的使用应遵循接口文档，不可臆断返回值的意义，否则造成逻辑错误。

示例：

```
void foo(const string& s) {
    if (s.find("bar")) {     // Non-compliant
        ....
    }
}
```

例中 find 函数返回 "bar" 在 s 中的位置，当 s 中不存在 "bar" 时返回 string::npos，将 find 函数的返回值转为 bool 型是没有逻辑意义的。

应改为：

```
void foo(const string& s) {
    if (s.find("bar") != string::npos) {    // Compliant
        ....
    }
}
```

想当然地认为返回 0 表示失败或不存在，非 0 表示成功或存在，是造成错误的常见原因。

又如：

```
bool gt(const char* a, const char* b) {
    return strcmp(a, b) == 1;  // Non-compliant
}
```

strcmp 函数的返回值可以是等于、大于或小于 0 的任意整数，分别对应字符串的等于、大于或小于关系，认为其只能返回 0、1 或 -1 是一种常见的误解。

应改为：

```
bool gt(const char* a, const char* b) {
    return strcmp(a, b) > 0;   // Compliant
}
```

strcmp、wcscmp 以及 memcmp 等函数不应与 0 之外的任何值比较。

下列函数的返回值不应与 0 比较，也不应转为 bool 型：

- open、create 等 Linux 系统调用，失败时返回负数，成功时返回非负数
- CreateFile、CreateNamedPipe 等 Windows API，失败时返回 INVALID_HANDLE_VALUE，而不是 0
- HRESULT 型 Windows API 返回值，负数表示失败、非负数表示成功

另外，有相当一部分函数成功时返回 0，失败时返回非 0，如 access、chmod、rename 等 Linux 系统调用，不可将其返回值当作 bool 型使用。

依据

ISO/IEC 9899:1999 7.21.4
ISO/IEC 9899:2011 7.24.4

参考

CWE-253

R10.4.3 避免对象切片

ID_objectSlicing　　expression warning

将派生类对象复制为基类对象的行为称为“对象切片（object slicing）”，基类对象不再持有派生类的属性，不再遵循多态机制，意味着某种精度上的损失，往往会造成意料之外的错误。

示例：

```
struct A { .... };
struct B: A { .... };

void foo(A);

A a;
B b;

a = b;   // Slicing
foo(b);  // Slicing
vector<A> v{b};  // Slicing
v.push_back(b);  // Slicing
```

尤其是函数传参或容器收纳对象时发生切片，会引起相当大的困惑，明明传入的是派生类对象，但虚函数都不生效了，所以要求多态性的接口或容器均应使用指针或引用。

在少数情况下，对象切片可能也有其逻辑意义，但不建议“隐式切片”，应定义特定名称的函数标明这是一种特殊处理，如：

```
A a;
B b;

a = b;   // Bad, implicit slicing
a = to_base_object(b);   // OK
```

其中 to_base_object 是一个返回基类对象的函数，表示有意为之。

相关

ID_paramMayBeSlicing

参考

C++ Core Guidelines ES.63
C++ Core Guidelines C.145
SEI CERT OOP51-CPP

R10.4.4 非基本类型的对象不应传入可变参数列表

ID_userObjectAsVariadicArgument　　expression error

非基本类型的对象与可变参数列表的机制很难相容，如果这种对象被传入可变参数列表，往往意味着错误。

传入可变参数列表的只应是基本类型的常量、变量、枚举值或指针，否则如果传入的对象存在自定义拷贝构造函数、移动构造函数或析构函数时，相关标准的定义较为粗略，往往是“conditionally-supported”或者“implementation-defined”。

示例：

```
struct A {
    string s;
    operator const char*() const {
        return s.c_str();
    }
};

void foo(const A& a) {
    printf("%s\n", a);  // Non-compliant
}
```

即使对象有转为 const char* 的方法，在可变参数列表中也是无效的，printf 无法正确获取字符串地址，造成内存访问错误。

应改为：

```
void foo(const A& a) {
    printf("%s\n", static_cast<const char*>(a));  // Compliant
}
```

依据

ISO/IEC 14882:2011 5.2.2(7)-implementation

参考

CWE-686
SEI CERT EXP47-C

R10.4.5 C 格式化字符串与其参数的个数应一致

ID_inconsistentFormatArgNum　　expression error

格式化字符串与其对应参数的个数应严格一致，否则会引发严重的运行时堆栈错误。

示例：

```
void foo(int type, const char* msg) {
    printf("Error (type %d): %s\n", type);  // Non-compliant
}
```

示例代码的格式化参数需要两个，但只传入一个，参数 msg 之外的不相关栈信息也会被读入。

由于可变参数列表自身的局限，很难在编译时发现这种问题，有些编译器会检查 printf、sprintf 等标准函数，但无法检查自定义函数，建议在 C++ 代码中禁用可变参数列表和 C 风格的格式化函数。

相关

ID_inconsistentFormatArgType
ID_forbidCStringFormat

依据

ISO/IEC 9899:2011 7.16.1.1(2)-undefined
ISO/IEC 9899:2011 7.21.6.1(2)-undefined

参考

SEI CERT FIO47-C

R10.4.6 C 格式化字符串与其参数的类型应一致

ID_inconsistentFormatArgType　　expression error

格式化字符串与其对应参数的类型应严格一致，否则会引发严重的运行时堆栈错误。

示例：

```
void foo(const string& msg) {
    printf("Message: %s\n", msg);  // Non-compliant
}
```

例中 %s 要求对应 char* 型参数，则 msg 实际上是 string 类型，造成栈读取错误。

应改为：

```
void foo(const string& msg) {
    cout << "Message: " << msg << '\n';  // Compliant
}
```

由于可变参数列表自身的局限，很难在编译时发现这种问题，有些编译器会检查 printf、sprintf 等标准函数，但无法检查自定义函数，建议在 C++ 代码中禁用可变参数列表和 C 风格的格式化函数。

相关

ID_userObjectAsVariadicArgument
ID_inconsistentFormatArgNum
ID_forbidCStringFormat

依据

ISO/IEC 9899:2011 7.16.1.1(2)-undefined
ISO/IEC 9899:2011 7.21.6.1(2)-undefined

参考

CWE-686
SEI CERT FIO47-C

R10.4.7 在 C++ 代码中禁用 C 风格字符串格式化方法

ID_forbidCStringFormat　　expression suggestion

---

printf、sprintf 等 C 风格字符串格式化方法，即由可变参数列表实现的格式化方法，主要问题有：

- 在编译期无法保证安全性，易出错或造成可移植性问题，提高了测试成本

- 与 C++ 的强类型理念不符，也不在 C++ 标准之内
- 只接受基本类型的参数，不利于数据的对象化管理

示例：

```
size_t a = -1;
ptrdiff_t b = -2;
```

如果要按 16 进制打印 a，10 进制打印 b：

```
printf("%x %d", a, b);  // Non-compliant, #1
printf("%lx %ld", a, b);  // Non-compliant, #2
printf("%llx %lld", a, b);  // Non-compliant, #3
```

size_t、ptrdiff_t 等类型是由实现定义的，标准没有规定其是否一定对应 unsigned int、long 或 long long 类型，而 %d、%lx、%llx 只对应 int、long、long long 类型，所以示例代码都是不合理的。#1 在 64 位环境中会丢失数据，#3 在 32 位环境中会造成参数栈读取错误，#2 只在某些环境下可以正常工作不具备可移值性。

在 C 语言中正确的做法是 a 对应 %zx，b 对应 %zd，如：

```
printf("%zx %zd", a, b);  // Non-compliant in C++, even if the result is correct
```

参数的类型与个数和占位符必须严格对应，否则就会导致标准未定义的错误，当参数较多时极易出错，利用 C++ iostream 可有效规避这些问题：

```
std::cout << std::hex << a << ' ' << std::dec << b;  // Compliant
```

然而当参数较多时，利用 iostream 的方式在形态上可能较为“松散”，其可读性可能不如 printf 等函数，对于这个问题可参见 ID_forbidVariadicFunction 中的示例，用“模板参数包”等更安全的方法实现 printf 函数的功能。另外，C++20 的“std::format”也提供了更多的格式化方法。

相关

ID_forbidVariadicFunction

依据

ISO/IEC 9899:2011 7.16.1.1(2)-undefined
ISO/IEC 9899:2011 7.21.6.1(2)-undefined

参考

C++ Core Guidelines SL.io.3

R10.4.8 不应显式调用析构函数

ID_explicitDtorCall　　expression suggestion

显式调用析构函数使对象在生命周期未结束的时候被析构，在逻辑上使人困惑，而且在对象生命周期结束时其析构函数仍会被调用，有可能造成资源重复释放的问题。

示例：

```
class A {
    int* p = new int[123];

public:
   ~A() {
        delete[] p;
    }
};

void fun() {
    A a;
    a.~A();  // Non-compliant, explicitly call the destructor
}            // ~A() twice called, crash...
```

例中对象 a 的析构函数被显式调用，之后 a 的生命周期结束会再次调用析构函数，造成内存的重复释放。应去掉显式调用，由类提供提前释放资源的方法，并保证资源不会被重复释放。

相关

ID_missingResetNull

R10.4.9 合理使用 std::move

ID_unsuitableMove　　expression warning

std::move 的参数应为左值，返回值应直接作为接口的参数，除此之外的使用方式价值有限，且易产生错误。

std::move 将左值转为右值，意在宣告对象的数据将被转移到其他对象，应由合适的接口完成数据转移。

示例：

```
string foo();
string s = move(foo());  // Non-compliant
```

例中 foo 函数返回的是右值，如果再调用 std::move 是多余的，应将 std::move 去掉。

又如：

```
string a("....");
string&& b = move(a);  // Non-compliant
string c(b);           // Not move construction
```

例中 b 是具有名称的右值引用，其实是左值，c 仍是拷贝构造。

应改为：

```
string a("....");
string c(move(a));  // Compliant
```

这样构造 c 时会自动选取移动构造函数，避免了复制。

又如：

```
string foo() {
    string s("....");
    ....
    return move(s);  // Non-compliant
}
```

例中 foo 函数返回对象，编译器会进行“RVO（Return Value Optimization）”优化，显式调用 move 是多余的，而且会干扰优化，不应出现 return std::move(....) 这种代码。

应改为：

```
string foo() {
    string s("....");
    ....
    return s;  // Compliant
}
```

参考

C++ Core Guidelines ES.56
C++ Core Guidelines F.18
C++ Core Guidelines F.48

R10.4.10 合理使用 std::forward

ID_unsuitableForward　　expression warning

---

std::forward 的参数应为“转发引用（forwarding references）”，返回值应直接作为接口的参数，除此之外的使用方式价值有限，且易产生错误。

转发引用是类型为 T&& 的参数，T 为函数模板类型，无论左值还是右值均可被这种参数接受，而且 const、volatile 等属性也会被忽略，这种参数应通过 std::forward 交由合适的接口处理。

关于转发引用，可参见 ID_illForwardingReference 的进一步说明。

示例：

```
struct A { .... };

void foo(A&);        // #1
void foo(const A&);  // #2
void foo(A&&);       // #3

template <class T>
void bar(T&& x) {    // Forwarding reference
    foo(forward<T>(x));  // Compliant
}

void baz(const A& a) {
    A b(a);
    bar(b);    // Calls #1

    bar(a);    // Calls #2

    bar(A());  // Calls #3
}
```

例中 bar 接口的参数为转发引用，在 baz 函数中，bar 接口将左值、常量引用和临时对象分别转发给对应的 foo 接口，这种模式称为“完美转发”，std::forward 应在这种模式内使用。

下面给出几种错误示例：

```
void bar(A&& x) {
    foo(forward<A>(x));  // Non-compliant, ‘x’ is not a forwarding reference
}

template <class T>
struct X {
    void bar(T&& x) {
        foo(forward<T>(x));  // Non-compliant, ‘x’ is not a forwarding reference
    }
};
```

注意，转发引用的类型只能是函数模板类型，非模板和类模板不构成转发引用。

```
template <class T>
void bar(T&& x) {
    T y = forward<T>(x);  // Non-compliant, ‘y’ is always an lvalue
    foo(y);
}

template <class T>
void bar(T&& x) {
    foo(forward<T&>(x));  // Non-compliant, use ‘forward<T>(x)’
}
```

forward 的返回值应直接作为接口的参数，且只应使用 forward。

相关

ID_illForwardingReference

参考

C++ Core Guidelines F.19

10.5 Sizeof

R10.5.1 sizeof 不应作用于有副作用的表达式

ID_sizeof_sideEffect　　expression warning

---

sizeof 只关注类型，其子表达式不会被求值，如果存在可以影响程序状态的运算符或函数调用，也不会有实际效果。

示例：

```
int a = 0;
cout << sizeof(a++) << ' ';  // Non-compliant
cout << a << '\n';  // What is output?
```

输出 4 0，a++ 不会被执行。

依据

ISO/IEC 14882:2003 5.3.3(1)
ISO/IEC 14882:2011 5.3.3(1)
ISO/IEC 14882:2017 8.3.3(1)

参考

SEI CERT EXP52-CPP
MISRA C 2004 12.3
MISRA C 2012 13.6
MISRA C++ 2008 5-3-4

R10.5.2 sizeof 的结果不应与 0 以及负数比较

ID_sizeof_zeroComparison　　expression error

标准规定，sizeof 的结果为无符号整型，对于完整类型结果一定不为 0，对于不完整类型则无法通过编译，所以将 sizeof 的结果与 0 甚至负数比较往往意味着逻辑错误。

示例：

```
void foo(int* p) {
    if (sizeof(p)) {  // Non-compliant, always true
        ....
    }
}
```

依据

ISO/IEC 14882:2003 5.3.3(1 6) 9(3)
ISO/IEC 14882:2011 5.3.3(1 6) 9(3)
ISO/IEC 14882:2017 8.3.3(1 6) 12(4)

参考

CWE-1025

R10.5.3 对数组参数不应使用 sizeof

ID_sizeof_arrayParameter　　expression error

当函数的形式参数为数组时，实际上是一个指针，对这种参数使用 sizeof 无法获取到数组大小，往往意味着错误。

示例：

```
void fun(char arr[32]) {
    memset(arr, 0, sizeof(arr));  // Non-compliant
}
```

例中参数 arr 是一个指针，而不是一个真实的数组。

如果有必要将参数设为数组，建议使用引用的方式，如：

```
void fun(char (&arr)[32]) {
    memset(arr, 0, sizeof(arr));  // Compliant
}
```

这样 sizeof(arr) 的结果就会和预期一致。

依据

ISO/IEC 9899:1999 6.7.5.3(7)
ISO/IEC 9899:2011 6.7.6.3(7)

参考

CWE-467

R10.5.4 sizeof 不应作用于逻辑表达式

ID_sizeof_oddExpression　　expression warning

---

sizeof 作用于 <、>、<=、>=、==、!=、&&、|| 等逻辑表达式为常见笔误，逻辑运算符往往应该移出 sizeof 表达式。

示例：

```
if (sizeof(a == x)) {  // Non-compliant, may be sizeof(a) == x
    ....
}
if (sizeof(a >= x)) {  // Non-compliant, may be sizeof(a) >= x
    ....
}
```

R10.5.5 被除数不应是作用于指针的 sizeof 表达式

ID_sizeof_pointerDivision　　expression warning

形如 sizeof(p)/n 的表达式往往是为了获取数组元素的个数，如果 p 是指针，sizeof(p) 只是指针变量的大小，并不是数组的大小，所以这种表达式往往意味着逻辑错误。

示例：

```
void foo(T* p) {
    size_t n = sizeof(p[0]);
    qsort(p, sizeof(p) / n, n, cmp);  // Non-compliant, logic error
}
```

例中 sizeof(p) / n 并不能获取 p 所指数组中元素的个数。

本规则是 ID_sizeof_pointer 的特化，sizeof 作用于指针是可疑的，如果这种表达式又作为被除数，就更加可疑了。

相关

ID_sizeof_pointer

参考

CWE-467

R10.5.6 指针加减偏移量时计入 sizeof 是可疑的

ID_sizeof_suspiciousAdd　　expression suspicious

指针加减偏移量时会自动计入指针指向类型的大小，如果再计入 sizeof 的值，很可能是某种错误。

示例：

```
int foo(int* p, int i) {
    return *(p + i * sizeof(int));  // Rather suspicious
}
```

如果 foo 函数是为了获取指针 p 之后第 i 个整数的值，那么这种实现是错误的，应改为：

```
int foo(int* p, int i) {
    return *(p + i);      // Right
}
```

依据

ISO/IEC 9899:1999 6.5.6(8)
ISO/IEC 9899:2011 6.5.6(8)

参考

CWE-468

R10.5.7 sizeof 不应再作用于 sizeof

ID_sizeof_sizeof　　expression error

sizeof(sizeof(....)) 等价于 sizeof(size_t)，在实际应用中没有任何必要写成连续 sizeof 的形式，属于常见笔误，多数由复制粘贴或错误的宏展开导致。

示例：

```
void foo() {
    auto* sa = (PSecAttr)LocalAlloc(LPTR, sizeof(PSecAttr));
    sa->nLength = sizeof(sizeof(PSecAttr));  // Non-compliant, copy-paste error
    ....
}
```

依据

ISO/IEC 14882:2003 5.3.3(6)
ISO/IEC 14882:2011 5.3.3(6)
ISO/IEC 14882:2017 8.3.3(6)

参考

CWE-682

R10.5.8 C++ 代码中 sizeof 不应作用于 NULL

ID_sizeof_NULL　　expression warning

在 C++ 语言中，标识符 NULL 并不能有效区分整型常量 0 和空指针，sizeof(NULL) 一类的表达式预期是获取指针变量的大小，而实际结果可能是整型变量的大小。

示例：

```
size_t n = sizeof(NULL);     // Non-compliant
```

不同的编译器对示例代码有不同的处理，有些会把 NULL 当作指针，有些会当作常量 0。

应改为：

```
size_t n = sizeof(nullptr);  // Compliant
```

相关

ID_deprecatedNULL

依据

ISO/IEC 9899:1999 7.17(3)-implementation
ISO/IEC 9899:2011 7.19(3)-implementation
ISO/IEC 14882:2003 C.2.2.3(1)-implementation
ISO/IEC 14882:2011 C.3.2.4(1)-implementation
ISO/IEC 14882:2017 C.5.2.7(1)-implementation

参考

CWE-351

R10.5.9 sizeof 不可作用于 void

ID_sizeof_void　　expression error

---

void 表示不存在的类型，也是不完整的类型，sizeof 作用于 void 是没意义的，属于语言运用错误，也可能是 sizeof(void*) 的笔误。

示例：

```
size_t a = sizeof(void);   // Non-compliant
size_t b = sizeof(void*);  // Compliant
```

依据

ISO/IEC 9899:1999 6.3.2.2(1)
ISO/IEC 9899:1999 6.2.5(19)

10.6 Assertion

R10.6.1 断言中的表达式不应恒为真

ID_badAssertion　　expression error

恒为真的断言是没有意义的。

示例：

```
void foo(int a[]) {
    assert(sizeof(a));  // Non-compliant
    assert("some comments");  // Non-compliant
}
```

也不建议使用恒为假的断言表示异常，在 C++ 语言中应改用异常处理的方式。

```
void bar(int x) {
    if (x < 0) {
        assert(0);  // Bad, use exceptions instead
    }
}
```

配置

names：断言函数或宏的名称，如 assert、_ASSERT_EXPR 等，用“|”分隔

依据

ISO/IEC 9899:2011 7.2

R10.6.2 断言中的表达式不应有副作用

ID_sideEffectAssertion　　expression error

断言中的表达式如果有副作用，不能保证在所有编译设置下都有效。

如标准断言 assert 会受宏 NDEBUG 的影响，当该宏被定义时 assert 中的表达式不会被执行。

示例：

```
void foo(int* p) {
    assert(++(*p) > 0);  // Non-compliant
}
```

表达式的副作用均应在 assert 之前完成：

```
void foo(int* p) {
    ++(*p);
    assert(*p > 0);      // Compliant
}
```

配置

names：断言函数或宏的名称，如 assert、_ASSERT_EXPR 等，用“|”分隔

相关

ID_macro_sideEffectArgs

依据

ISO/IEC 9899:2011 7.2

参考

SEI CERT PRE31-C

R10.6.3 断言中的表达式不应过于复杂

ID_complexAssertion　　expression suggestion

断言中的表达式不应过于复杂， 否则不易定位具体是哪一项不符合断言，不利于调试。

对于“逻辑与”表达式应将各子表达式分成多个断言。

示例：

```
void foo(int a, int b, int c) {
    assert(a != 0 && b > 10 && c == b + 1);  // Bad
}
```

应改为：

```
void foo(int a, int b, int c) {
    assert(a != 0);
    assert(b > 10);
    assert(c == b + 1);  // Good
}
```

本着使代码便于调试的理念展开工作，可有效降低测试及维护成本。

配置

maxLogicOperatorCount：断言表达式中“逻辑与”运算符的最大数量，超过则报出
names：断言函数或宏的名称，如 assert、_ASSERT_EXPR 等，用“|”分隔

10.7 Complexity

R10.7.1 运算符不应超过规定数量

ID_complexExpression　　expression suggestion

运算符超过一定数量意味着表达式过于复杂，易包含潜在的错误，更不利于调试与维护，应进行适当拆分。

配置

maxLogicOperatorCount：逻辑运算符最大数量，超过则报出
maxOperatorCount：运算符最大数量，超过则报出

参考

C++ Core Guidelines ES.40

10.8 Other

R10.8.1 不应访问填充数据

ID_accessPaddingData　　expression warning

变量之间可能存在填充数据，这种数据只为实现“内存对齐”而无数值意义，而且填充数据的值是标准未声明的。

示例：

```
struct A {
    char a;
    long b;
};

void foo(A* x, A* y) {
    if (memcmp(x, y, sizeof(*x)) == 0) {  // Non-compliant
        ....
    }
}
```

如果按常见的 4 或 8 字节对齐，A 的成员 a 和 b 之间会存在填充数据，填充数据参与比较将得到错误的结果。

应改为：

```
void foo(A* x, A* y) {
    if (x->a == y->a && x->b == y->b) {  // Compliant
        ....
    }
}
```

即使成员之间没有填充数据也不应使用 memcmp 等函数比较，应实现类型明确的比较接口以供调用，否则很容易产生错误。

依据

ISO/IEC 9899:2011 6.2.6.2(5)-unspecified

R10.8.2 new 表达式只可用于赋值或当作参数

ID_oddNew　　expression warning

---

new 表达式只应作为“=”的直接右子表达式，或直接作为参数表达式，其他形式均有问题。

本规则对 replacement-new 暂不作要求。

示例：

```
int& i = *new int(123);        // Non-compliant

if (new int[123]) {            // Non-compliant
    ....
}

char* p = new char[123] + n;   // Non-compliant
```

这些问题多数是笔误或错误的宏展开造成的。

相关

ID_multiAllocation

R10.8.3 数组下标应为整形表达式

ID_oddSubscripting　　expression warning

C/C++ 语言规定，数组下标可以在中括号的右侧也可以在左侧，然而这只是一种理论上的设计，在实际代码中，应采用约定俗成的方式，即数组的名称在中括号的左侧，下标在中括号的右侧。

示例：

```
void fun() {
    int a[5] = {1, 2, 3, 4};
    4[a] = 5;                       // Non-compliant, use a[4] instead
    for (int i = 0; i < 4; i++) {
        cout << i + 1[a] << '\n';  // Non-compliant, may be a[i + 1]
    }
}
```

由于 a[4] 相当于 *(a+4)，所以与其等价的 *(4+a) 也应该可以解释为 4[a]，但下标在左侧的写法容易造成运算符优先级相关的问题，将下标写在右侧才是符合惯例的方式。

依据

ISO/IEC 14882:2003 8.3.4(6)
ISO/IEC 14882:2011 8.3.4(6)

R10.8.4 禁用逗号表达式

ID_forbidCommaExpression　　expression suggestion

逗号表达式将多个语句合成一个表达式，易形成笔误并造成阅读困难。

示例：

```
a = b++, b + 1;       // Non-compliant
a, b, c = 0, 1, 2;    // Non-compliant
delete p, q;          // Non-compliant
foo((a, b), c);       // Non-compliant
return a, b, c;       // Non-compliant
```

逗号运算符和其他运算符组合在一起，易造成各种错误。

在 for 迭代声明中的第 1 个和第 3 个表达式中使用逗号表达式为惯用方式，但这种方式并不值得提倡，不妨根据配置项选择是否放过这种情况。

```
for (a = 0, b = 0; a < 100; a++, b++)  {  // let it go?
    ....
}
```

配置

allowCommaExpressionInForIteration：为 true 时放过 for 迭代声明中的逗号表达式

参考

MISRA C 2004 12.10
MISRA C 2012 12.3
MISRA C++ 2008 5-18-1

R10.8.5 合理使用括号

ID_redundantParentheses　　expression suggestion

重复的、作用于单个对象或一元运算符的括号使代码显得繁琐，应去掉。

宏定义中的括号不受本规则限制。

示例：

```
a = 1 + (p[0]);      // Non-compliant
a = 2 + (p->n);      // Non-compliant
a = (sizeof(b));     // Non-compliant
a = (fun(b));        // Non-compliant
a = ((a + b)) * b;   // Non-compliant
```

应去掉多余的括号：

```
a = 1 + p[0];        // Compliant
a = 2 + p->n;        // Compliant
a = sizeof(b);       // Compliant
a = fun(b);          // Compliant
a = (a + b) * b;     // Compliant
```

但如果可以更好的表达逻辑意义，或不确定运算符优先级时，应及时使用括号，如：

```
if (a == b && b == c || x == y && y == z) {  // Bad
    ....
}
```

设例中两个 && 表达式对应两个逻辑步骤，虽然有无括号不影响计算结果，但建议加入括号提高可读性：

```
if ((a == b && b == c) || (x == y && y == z)) {  // Good
    ....
}
```

11. Literal

R11.1 注意可疑的字符常量

ID_literal_suspiciousChar literal warning

注意字符常量的错误书写，如正反斜杠的误用，'\n' 误写为 '/n'、'\\' 误写为 '//' 等。

由于 C/C++ 语言允许在单引号内写入多个字符来表示一个整形常量，如：

```
auto i = '/t';
cout << typeid(i).name() << ' ' << i;  // What is output?
```

例中 i 为 int 型变量，值为 12148，这种语言特性可以让一些笔误通过编译，造成不易察觉的问题。

又如：

```
auto* tab = wcschr(str, L'/t');
```

在某些环境中执行结果和下列代码一样：

```
auto* tab = wcschr(str, L'/');
```

字符 t 将被忽略，这显然是错误的，L'/t' 应改为 L'\t'。

相关

ID_literal_multicharacter

依据

ISO/IEC 14882:2011 2.13.2(1)-implementation
ISO/IEC 14882:2011 2.14.3(1)-implementation
ISO/IEC 14882:2017 5.13.3(2)-implementation

R11.2 字符常量中不可存在应转义而未转义的字符

ID_literal_hardCodeChar literal warning

在字符常量中，如果存在制表符或控制字符，应使用转义字符。

示例：

```
char c = '  ';  // Non-compliant
```

例中空白符为 tab，由于 tab 可以按不同的宽度被展示，往往会被代码的阅读者误解为空格，而且经过复制粘贴后有可能会变成数目不确定的空格，使代码难以维护。

应使用转义字符：

```
char c = '\t';  // Compliant
```

相关

ID_literal_hardCodeString

R11.3 字符串常量中不可存在应转义而未转义的字符

ID_literal_hardCodeString　　literal warning

在字符串常量中，如果存在制表符或控制字符，应使用转义字符。

示例：

```
const char* s = "a  b";  // Non-compliant
```

例中空白符为 tab，由于 tab 可以按不同的宽度被展示，往往会被代码的阅读者误解为空格，而且经过复制粘贴后有可能会变成数目不确定的空格，使代码难以维护。

应使用转义字符：

```
const char* s = "a\tb";  // Compliant
```

相关

ID_literal_hardCodeChar

R11.4 不应使用非标准转义字符

ID_literal_nonStandardEsc　　literal warning

使用非标准转义字符会导致标准未定义的问题，也可能是忘了将反斜杠转义。

示例：

```
string path("C:\Files\x.cpp");  // Non-compliant
```

例中 \F 不是标准转义字符，\x 也不符合 16 进制转义字符的格式，这显然是路径中的反斜杠忘了转义。

附 C/C++ 标准转义字符：

```
'\a'      // Alert
'\b'      // Backspace
'\f'      // Formfeed page break
'\n'      // New line
'\r'      // Carriage return
'\t'      // Horizontal tab
'\v'      // Vertical tab
'\\'      // Backslash
'\?'      // Question mark
'\''      // Single quotation mark
'\"'      // Double quotation mark
'\0'      // Null character
'\ddd'    // Any character, d is an octal number
'\xhh'    // Any character, h is a hex number
```

依据

ISO/IEC 14882:2003 2.13.2(3)-undefined
ISO/IEC 14882:2011 2.14.3(3)-implementation
ISO/IEC 14882:2017 5.13.3(7)-implementation

参考

MISRA C 2004 4.1
MISRA C++ 2008 2-13-1

R11.5 不同前缀的字符串常量不可连接在一起

ID_literal_hybridConcat　　literal warning

L、U、u、u8 等字符串前缀表示不同的类型，不可连接在一起。

示例：

```
auto* a = L"123" U"456";  // Non-compliant
auto* b = U"123" u"456";  // Non-compliant
```

应保持一致：

```
auto* a = L"123" L"456";  // Compliant
auto* b = U"123" U"456";  // Compliant
```

C++03 规定宽字符串与窄字符串连接是未定义的。
C++11 规定一个字符串有前缀一个没有的话，结果以有前缀的为准，其他情况由实现定义。

如：

```
auto* x = L"123" "456";    // Undefined in C++03
auto* y = L"123" "456";    // A wide string in C++11
auto* z = L"123" u8"456";  // Ill-formed in C++11
```

依据

ISO/IEC 14882:2003 2.13.4(3)-undefined
ISO/IEC 14882:2011 2.14.5(13)-implementation

参考

MISRA C++ 2008 2-13-5

R11.6 字符串常量中不应存在拼写错误

ID_literal_misspelling literal warning

如果含有拼写错误的常量字符串对用户可见，也可以认为是产品的一种 bug，会对用户造成困扰，故应认真对待。

示例：

```
void showMessage(int errCode) {
    if (errCode) {
        cout << "Error\n";
    } else {
        cout << "Successfull\n";  // Non-compliant
    }
}
```

例中“Successfull”存在拼写错误，应改为“Successful”。

相关

ID_misspelling
ID_macro_misspelling

R11.7 整数或浮点数常量的后缀应使用大写字母

ID_literal_confusingSuffix literal warning

整数或浮点数常量的后缀应使用大写字母，否则小写字母“l”极易与数字“1”混淆。

示例：

```
long long a = 100ll;  // Non-compliant, misread as 1001l
long double b = 100.l;  // Non-compliant, misread as 100.1
```

应改为：

```
long long a = 100LL;  // Compliant
long double b = 100.L;  // Compliant
```

后缀大小写混用的情况会使人更加困惑：

```
long long c = 100lL;  // Non-compliant, very bad
unsigned long long d = 100lLU;  // Non-compliant, very bad
```

其中，小写的 l 和大写的 L 混在了一起。

应改为：

```
long long c = 100LL;  // Compliant
unsigned long long d = 100LLU;  // Compliant
```

参考

C++ Core Guidelines NL.19
MISRA C 2012 7.3
MISRA C++ 2008 2-13-4

R11.8 禁用 8 进制常量

ID_literal_forbidOct literal suggestion

---

8 进制不像 10 进制那样符合人们的常规思维，也不像 2 进制或 16 进制那样便于展示数据的存储格式，而且 C/C++ 中 8 进制表示法只是在数字前置 0，与十进制过于相似，易被误用。

示例：

```
const int K_0 = 5592;
const int K_1 = 0631;  // Non-compliant
const int K_2 = 3817;
const int K_3 = 4257;
```

为了格式上的对齐，错误地在 10 进制数前写 0 是常见笔误，例中 k_1 的实际值为 409。

参考

MISRA C 2004 7.1
MISRA C 2012 7.1
MISRA C++ 2008 2-13-2

R11.9 整数或浮点数常量应使用标准后缀

ID_literal_nonStandardSuffix　　literal suggestion

整数常量后缀只应为 L、LL、UL、ULL，浮点数常量的后缀只应为 L、f 或 F，其他非标准后缀没有可移植性。

示例：

```
unsigned int a = 100ui32;  // Non-compliant, not common between compilers
long long b = 100i64;      // Non-compliant
```

应改为：

```
unsigned int a = 100U;  // Compliant
long long b = 100LL;    // Compliant
```

相关

ID_literal_confusingSuffix

依据

ISO/IEC 14882:2003 2.14.2(2) 2.14.4(1)
ISO/IEC 14882:2011 2.14.2(2) 2.14.4(1)
ISO/IEC 14882:2017 5.13.2(2) 5.13.4(1)

R11.10 小心遗漏逗号导致的非预期字符串连接

ID_literal_oddConcat literal warning

注意可能导致非预期结果的字符串连接，尤其在初始化列表中，小心逗号被遗漏。

示例：

```
string strs[] = {
    "123", "456", "789",
    "123", "456", "789"   // Rather suspicious, missing a comma?
    "123", "456", "789"
};
```

例中初始化列表第 2 行的 "789" 与第 3 行的 "123" 中间没有逗号，会连接成 "789123"，显然是不符合预期的，这种问题属于常见笔误。

又如：

```
void foo(const char*);
void foo(const char*, const char*);

void bar() {
    foo("abc" "123");  // Rather suspicious, which ‘foo’ is right?
}
```

常量字符串连接的用途：

- 字符串过长不便于显示时可将字符串拆成多个子串分行书写
- 宏和字符串连接在一起完成一些更灵活的操作

除此之外不应再将一个字符串拆成多个常量。

R11.11 不应存在 magic number

ID_literal_magicNumber literal suggestion

对于直接出现在代码中的字面数值（magic number），建议改用具有适当名称的常量或枚举项表示。

示例：

```
for (int i = 0; i < 12345; i++) {  // Non-compliant, 12345 is a magic number
    ....
}
```

例中 12345 不能表示其具体的含义，而且当这种 magic number 散落在代码的各个角落时，不便于统一管理，造成维护上的困难。

应改为具有名称的常量：

```
const int maxId = 12345;

for (int i = 0; i < maxId; i++) {  // Compliant
    ....
}
```

配置

magicNumberDigitThreshold：数字常量的位数上限，超过则报出

相关

ID_literal_magicString

参考

C++ Core Guidelines ES.45

R11.12 不应存在 magic string

ID_literal_magicString　　literal suggestion

对于直接出现在代码中的字符串常量（magic string），建议改用具有适当名称的常量表示。

示例：

```
void foo(const string& s) {
    if (s == "https://foo.net/bar") {  // Non-compliant
        bar("https://foo.net/bar");    // Non-compliant
    }
}
```

当这种常量字符串散落在代码的各个角落时，不便于统一管理，造成维护上的困难。

应改为：

```
const char url[] = "https://foo.net/bar";  // Compliant

void foo(const string& s) {
    if (s == url) {
        bar(url);
    }
}
```

相关

ID_literal_magicNumber

参考

C++ Core Guidelines ES.45

R11.13 不应使用多字符常量

ID_literal_multicharacter literal suggestion

多字符常量形式上与字符串常量相似，但类型为整型，易被误用，而且不同编译器对这种常量的处理方式也有所不同，故建议禁用。

示例：

```
void foo(int x) {
    if (x == 'tcp') {  // Non-compliant
        ....
    }
    else if (x == 'udp') {  // Non-compliant
        ....
    }
}
```

例中 'tcp'、'udp' 为多字符常量，应改用普通常量。

本例在 C++ 中也可使用 enum class 实现：

```
enum class PROT { tcp, udp };

void foo(PROT x) {
    if (x == PROT::tcp) {  // Compliant
        ....
    }
    else if (x == PROT::udp) {  // Compliant
        ....
    }
}
```

相关

ID_literal_suspiciousChar

依据

ISO/IEC 14882:2011 2.13.2(1)-implementation
ISO/IEC 14882:2011 2.14.3(1)-implementation
ISO/IEC 14882:2017 5.13.3(2)-implementation

12. Cast

R12.1 避免类型转换造成数据丢失

ID_narrowCast　　cast warning

类型转换时应检查转换的结果是否正确，避免数据丢失等错误。

示例：

```
void foo(int i) {
    char a = i;        // Non-compliant, check it first
    char b = (char)i;  // Non-compliant, irresponsible
    ....
}
```

下面给出判断转换后数据是否完整的简单示例：

```
template <class To, class From>
To checked_cast(From x) noexcept(false) {
    auto y = static_cast<To>(x);
    auto z = static_cast<From>(y);
    return x == z? y: throw DataLoss();
}

void foo(int i) {
    char a = checked_cast<char>(i);  // Compliant
    ....
}
```

例中模板函数 checked_cast 将源类型转为目标类型，再将目标类型转回源类型，如果经两次转换得到的数据与源数据不符，说明转换存在数据丢失，抛出相关异常。

在实际代码中还需要考虑负数与无符号数、浮点数与整数之间的转换是否存在逻辑意义。

参考

C++ Core Guidelines ES.46

R12.2 避免向下类型转换

ID_downCast　　cast suggestion

当代码中出现了从基类到派生类的向下类型转换，以及可以造成数据损失的类型转换，意味着现有接口或流程不能满足需求，需要“特殊处理”，所以这种转换越少越好。

示例：

```
class A {
    ....
};

class B: public A {
    ....
public:
    void bar();  // A new function
};

void foo(A* a) {
    ....
    if (auto* p = dynamic_cast<B*>(a)) {  // Bad
        p->bar();
    }
    ....
}
```

例中 foo 接口对 B 类型进行特殊处理，是不利于维护的，当这种特殊处理较多时，应利用虚函数、重载或模板等方法进行合理重构。

```
class A {
    ....
public:
    virtual void bar();
};

class B: public A {
    ....
public:
    void bar() override;
};

void foo(A* a) {
    ....
    if (a) {
        a->bar();  // Good
    }
    ....
}
```

相关

ID_nonDynamicDownCast
ID_narrowCast

参考

C++ Core Guidelines ES.48

R12.3 指针与整数不应相互转换

ID_ptrIntCast　　cast warning

指针与整数相互转换，容易造成地址不完整、寻址错误、降低可移植性等多种问题。

指针与整数的转换由实现定义，整数的符号和取值范围可能与指针有冲突，错误的值转为指针也会导致标准未定义的行为。

示例：

```
void foo(int* p) {
    vector<int> v;
    v.emplace_back((int)p);  // Non-compliant
    ....
}
```

例中将指针 p 转为 int 是不符合要求的，指针的值可能会超过 int 的范围。

在多数实现中，指针的值可以安全地转为 size_t 类型，不妨通过配置决定是否放过指针与 size_t 的转换。

配置

allowPointerToSizeType：为 true 时可以放过指针与 size_t 的转换

相关

ID_fixedAddrToPointer

依据

ISO/IEC 9899:2011 6.3.2.3(5)-implementation
ISO/IEC 14882:2011 5.2.10(4 5)-implementation

参考

SEI CERT INT36-C
MISRA C 2004 11.3
MISRA C 2012 11.4
MISRA C++ 2008 5-2-8
MISRA C++ 2008 5-2-9

R12.4 类型转换时不应去掉 const、volatile 等属性

ID_qualifierCastedAway cast warning

类型转换时去掉 const、volatile 等属性使相关机制失去了意义，这往往意味着设计上的缺陷，也会导致标准未定义的错误。

示例：

```
const int N = 123;

void foo(const int& n) {
    const_cast<int&>(n) = 456;   // Non-compliant
}

void bar() {
    foo(N);  // Undefined behavior, modifies a const object
}
```

依据

ISO/IEC 14882:2003 7.1.5.1(4 5)-undefined
ISO/IEC 14882:2011 7.1.6.1(4)-undefined

参考

C++ Core Guidelines Type.3
MISRA C 2004 11.5
MISRA C 2012 11.8
MISRA C++ 2008 5-2-5

R12.5 不应强制转换无继承关系的类型

ID_castNoInheritance　　cast warning

无继承关系的类型之间没有逻辑关系，不应强制转换，否则意味着设计缺陷或逻辑错误。

示例：

```
class A { .... };
class B { .... };
class C: public A, public B { .... };

void foo() {
    A* a = new C;

    B* b0 = (B*)a;                     // Non-compliant
    B* b1 = reinterpret_cast<B*>(a);  // Non-compliant

    B* b2 = static_cast<B*>(a);       // Compliant, compile-time protected
    B* b3 = dynamic_cast<B*>(a);      // Compliant, run-time protected

    ....
}
```

例中 A 与 B 没有继承关系，C 从 A 和 B 继承，指针 a 为 A 类型但实际指向 C 的实例，这种情况下将 a 直接强制转为 B 类型的指针将得到错误的结果，这个问题在实际代码中也很常见。

本规则限制无继承关系的 C 风格类型转换以及 reinterpret_cast 转换，放过 static_cast 和 dynamic_cast 转换，示例中的 static_cast 转换将得到编译错误从而锁定问题，如果 A 和 B 是多态类型，用 dynamic_cast 会得到正确的结果。

例外：

```
class V { .... };

class U {
    ....
public:
    operator V*();
};

void bar(U* u) {
    V* v0 = (V*)u;                     // Compliant, but bad
    V* v1 = reinterpret_cast<V*>(u);  // Still non-compliant
    ....
}
```

例中 U 和 V 是无继承关系的类，但 U 实现了向 V 的转换方法，U 和 V 之间存在逻辑关系，这时的 C 风格类型转换可被本规则放过，但不符合规则 ID_forbidCStyleCast，这种情况仍然不能使用 reinterpret_cast，可参见 ID_unsuitableReinterpretCast。

注意，对于基本类型，指针或引用之间的转换受本规则限制，值之间的转换可被放过，但应注意转换造成的精度损失，可参见 ID_narrowCast。

相关

ID_narrowCast
ID_unsuitableReinterpretCast

依据

ISO/IEC 14882:2003 5.2.10(7)-unspecified
ISO/IEC 14882:2011 5.2.10(7)-unspecified

参考

MISRA C 2012 11.3
MISRA C++ 2008 5-2-7

R12.6 不应强制转换非公有继承关系的类型

ID_castNonPublicInheritance　　cast warning

公有继承表示派生类是基类的某种扩展，而非公有继承往往表示派生类是基类的某种“例外”，基类的方法不再适用于派生类的对象。

示例：

```
class A { .... };
class B: private A { .... };

void bar(A* a);

void foo(B* b) {
    bar((A*)b);  // Non-compliant
}
```

例中 B 是对 A 的某种改造，如果再用 A 的方法去处理 B 的对象，显然是有问题的。

依据

ISO/IEC 9899:2011 4.10(3)

R12.7 多态类型与基本类型不应相互转换

ID_castViolatePolymorphism　　cast warning

多态类型会维护虚表指针等用户不可见的数据以保证多态机制的执行，将其与基本类型转换会破坏这种机制。

示例：

```
class A {
    ....

public:
    virtual ~A() = 0;

    void save() const {
        FILE* fp = fopen("dat", "wb");
        fwrite(this, sizeof(A), 1, fp);   // Non-compliant
        fclose(fp);
    }

    void load() {
        FILE* fp = fopen("dat", "rb");
        fread(this, sizeof(A), 1, fp);    // Non-compliant
        fclose(fp);
    }
};
```

例中 A 是多态类型，save 函数将对象写入文件，fwrite 的第一个参数 this 被隐式转为 void*，不符合本规则要求。对象的虚表指针等数据一并被写入文件，但虚表指针是运行时数据不应被保存，load 函数从文件中读取对象便破坏了运行时数据。

注意，正常的代码不应显式访问虚表指针等运行时数据，否则是不可移植的，标准只定义了多态类的行为，并未规定具体实现方式。

参考

CWE-843

R12.8 不可直接转换不同的字符串类型

ID_charWCharCast　　cast warning

---

char* 和 wchar_t* 直接转换并不进行字符集转换，属于语言运用错误，char*、wchar_t*、char16_t* 以及 char32_t* 之间均不可直接转换。

本规则是 ID_castNoInheritance 的特化。

示例：

```
wchar_t* to_unicode(char* str) {
    return (wchar_t*)str;  // Remarkably brave, but totally wrong
}
```

示例代码显然是错误的，应改用 iconv、MultiByteToWideChar 等字符集编码转换函数。

由于 unsigned char* 一般针对二进制数据，unsigned char* 与其他字符串类型之间的转换可被放过，但 char* 不应作为二进制数据的类型，可参见 ID_plainBinaryChar。

相关

ID_castNoInheritance
ID_plainBinaryChar

参考

CWE-704
SEI CERT STR38-C

R12.9 避免类型转换造成的指针运算错误

ID_arrayPointerCast　　cast warning

指针的逻辑大小与类型有关，不适当的类型转换会造成指针运算错误，应避免转换指向数组的指针。

示例：

```
struct A { int x; };
struct B: A { int y; };

void foo(A* p, int n) {
    for (int i = 0; i < n; i++) {
        p[i].x = 123;
    }
}

void bar() {
    B arr[10];
    foo(arr, 10);  // Bug
    ....
}
```

例中 B 类型的数组 arr 作为 foo 函数的参数，被转换成了基类指针，foo 函数中对基类指针的运算将是错误的，B 的成员 y 也会被错误地赋值。

这是一个很危险的问题，本规则针对所有数组相关的隐式和显式类型转换。

依据

ISO/IEC 9899:1999 6.5.6(8)
ISO/IEC 9899:2011 6.5.6(8)

参考

C++ Core Guidelines C.152

R12.10 对函数指针不应进行类型转换

ID_functionPointerCast　　cast warning

函数指针和不兼容的类型转换会导致标准未定义的行为。

示例：

```
void foo();
typedef void (*fnp_t)(int);

void* p0 = (void*)&foo;    // Non-compliant
fnp_t p1 = (fnp_t)&foo;    // Non-compliant

p1(123);                   // Undefined behavior
```

例外：

```
fnp_t p = NULL;            // Compliant

(void)p;                   // Let it go
p = (fnp_t)dlsym(h, "f");  // Let it go
```

对函数指针进行 (void) 转换可被放过，dlsym、GetProcAddress 等动态导入函数的系统接口可被放过。

依据

ISO/IEC 9899:2011 6.3.2.3(6 7 8)-undefined
ISO/IEC 9899:2011 6.5.2.2(9)-undefined
ISO/IEC 14882:2011 5.2.10(6)-undefined
ISO/IEC 14882:2011 5.2.10(8)-implementation

参考

MISRA C 2004 11.1
MISRA C 2012 11.1
MISRA C++ 2008 5-2-6

R12.11 向下类型转换应使用 dynamic_cast

ID_nonDynamicDownCast　　cast warning

向下类型转换如果不用 dynamic_cast 难以保证安全性。

示例：

```
class A { .... };
class B: public A { .... };

void foo(A* a) {
    bar((B*)a);  // Non-compliant
    baz(dynamic_cast<B*>(a));  // Compliant
}
```

如果参数 a 实际指向的不是 B 类的对象，(B*)a 将得到一个无法判断对错的值，而 dynamic_cast<B*>(a) 会得到一个空值，便于进一步处理。

注意，虚基类指针只能通过 dynamic_cast 转换为派生类指针，否则导致标准未定义的行为：

```
struct A { .... };
struct B: virtual A { .... };
struct C: virtual A { .... };
struct D: B, C { .... };

void foo(A* a) {
    D* d0 = (D*)a;  // Undefined behavior
    D* d1 = dynamic_cast<D*>(a);  // Right
    ....
}
```

应尽量减少向下类型转换，可参见 ID_downCast 的进一步讨论。

相关

ID_downCast

依据

ISO/IEC 14882:2011 5.2.7
ISO/IEC 14882:2003 5.2.9(5 8)-undefined
ISO/IEC 14882:2011 5.2.9(11 12)-undefined

参考

MISRA C++ 2008 5-2-2
C++ Core Guidelines Type.2

R12.12 对 new 表达式不应进行类型转换

ID_oddNewCast　　cast warning

---

new 表达式本身是类型明确的，转换 new 表达式的类型不符合 C++ 严谨的类型理念，也容易造成分配、访问或回收相关的错误。

示例：

```
int* p = (int*)new char[123];  // Non-compliant
....
delete[] p;  // What will happen?
```

例中 char 数组转为 int 数组，由于元素个数不兼容也会导致内存访问与回收的错误。

相关

ID_forbidFlexibleArray

R12.13 不应存在多余的类型转换

ID_redundantCast　　cast warning

转换前后的类型完全相同是没有意义的，很可能意味着某种笔误。

示例：

```
float foo(int a) {
    return (int)a;    // Non-compliant
}
```

应改为：

```
float foo(int a) {
    return (float)a;  // Compliant
}
```

参考

CWE-704

R12.14 可用 static_cast、dynamic_cast 完成的类型转换不应使用 reinterpret_cast

ID_unsuitableReinterpretCast　　cast warning

---

reinterpret_cast 将某地址强行按另一种类型解释，不考虑转换需要的逻辑，可用其他方法转换时不应使用 reinterpret_cast。

示例：

```
struct A {
    int a = 1;
};

struct B {
    int b = 2;
};

struct C: A, B {
};

int main() {
    C c;
    cout << static_cast<B*>(&c)->b << ' ';
    cout << reinterpret_cast<B*>(&c)->b << '\n';  // Non-compliant, what is
output?
}
```

输出 2 1，如果想将派生类的地址 &c 转为基类指针，应使用 static_cast 进行正确的偏移转换，而如果使用 reinterpret_cast，不会进行偏移转换，这种情况下得到的成员 b 是错误的。

依据

ISO/IEC 14882:2003 5.2.10(7)
ISO/IEC 14882:2011 5.2.10(7)

参考

C++ Core Guidelines Type.1

R12.15 在 C++ 代码中禁用 C 风格类型转换

ID_forbidCStyleCast　　⛔ cast suggestion

C 语言的类型观念弱于 C++，易造成逻辑错误或数据丢失，应尽量避免类型转换，或使用 static_cast、dynamic_cast 等方法。

示例：

```
class A { .... };
class B { .... };

void foo(A* a) {
    B* b = (B*)a;  // Non-compliant, an error value with no logical meaning
    ....
}

void bar(A* a) {
    B* b = dynamic_cast<B*>(a);  // Compliant, prevent errors at compile time
    ....
}
```

例中 A 和 B 是两种不相关的类型，用 C 语言的转换方式是可以转换成功的，但并没有逻辑意义，在 C++ 语言中应使用 static_cast 或 dynamic_cast 等方法在编译时或运行时保障转换的有效性。

参考

MISRA C++ 2008 5-2-4
C++ Core Guidelines ES.49

R12.16 合理使用 reinterpret_cast

ID_forbidReinterpretCast　　⛔ cast suggestion

语言对 reinterpret_cast 的定位不是为了安全性，而是为了灵活性，可以用其他方式实现的功能不应使用 reinterpret_cast，如果必须使用需提供明确的文档说明。

示例：

```
class MyData { .... };

void foo(const char* path) {
    unsigned char* p = read_from_file(path);
    MyData* dat = reinterpret_cast<MyData*>(p);  // Bad
    ....
}
```

例中通过 reinterpret_cast 将二进制数据直接转为 MyData 对象，这不是一种安全的方式，妥善的做法是根据文件数据将 MyData 的成员逐一构造出来，这样也可及时发现并处理问题。

又如：

```
struct data_type {
    void* dummy;
};

data_type* external_interface();  // If it's out of control ...

void foo() {
    auto* data = external_interface();
    auto* mydata = reinterpret_cast<MyData*>(data);  // Attention
    ....
}
```

例中 external_interface 是项目外部的一个接口，它的实现方式完全不受控制，返回类型也不是有效类型，甚至需要某种“hacking”才能使用这个接口，这已经不属于正常的开发范围了，可以用 reinterpret_cast 强调这是一种非正常的转换，但需注明这种情况产生的原因，以及是否有改进的余地等信息。

相关

ID_forbidCStyleCast

参考

CWE-843
C++ Core Guidelines Pro.safety

13. Buffer

R13.1 对缓冲区的读写应在有效边界内进行

ID_bufferOverflow　　buffer warning

“缓冲区（buffer）”的本意是指内存等高速设备上的区域，程序在这种区域内接收或处理数据，之后再一并输出到网络或磁盘等低速环境，起到提高效率的作用，故称缓冲区。连续的内存区域均可称为缓冲区，在 C/C++ 语言中对应数组等结构。

缓冲区之外可能是程序的其他数据，也可能是函数返回地址、资源分配信息等重要数据，对缓冲区的越界读写往往意味着逻辑错误，而且会使程序遭到破坏。

示例：

```
void foo(const char* s) {
    char buf[100];
    strcpy(buf, s);   // Non-compliant
    ....
}

int main() {
    foo(userInput());
}
```

例中 userInput 函数返回用户输入的字符串，其长度不确定，而缓冲区 buf 的长度为 100 字节，如果用户输入超过这个长度就会使程序遭到破坏，这种问题称为“缓冲区溢出（buffer overflow）”，也是程序遭受攻击的常见原因。

利用缓冲区溢出可造成严重危害，如：

- 破坏堆栈或段结构，扰乱程序执行
- 改写关键信息，篡改程序行为
- 注入并运行恶意代码
- 攻击高权限进程获取非法权限

所以将读写限定在缓冲区边界之内是十分重要的，示例代码应改为：

```
void foo(const char* s) {
    char buf[100] = "";
    strncpy(buf, s, sizeof(buf) - 1);   // Compliant
    ....
}
```

strncpy 与 strcpy 不同，当源字符串长度超过指定限制时会结束复制，但要注意 strncpy 对空字符的处理。

对数组下标越界的问题，本规则特化为 ID_arrayIndexOverflow。

相关

ID_arrayIndexOverflow
ID_unsafeStringFunction

参考

CWE-119
CWE-131
CWE-788

R13.2 数组下标不可越界

ID_arrayIndexOverflow　　buffer error

数组下标不在数组声明的大小范围之内，意味着内存读写错误，会导致难以控制的后果。

示例：

```
int foo() {
    int a[10] = {};
    ....
    return a[10];  // Non-compliant, overflow
}
```

例中数组声明的大小是 10，其下标有效范围是 0 至 9，10 不能再作为下标，这也是常见笔误。

如果数组下标可受用户控制，更应该判断是否在有效范围内，如：

```
const char* fruits[] = {
    "apple", "banana", "grape"
};

const char* bar() {
    return fruits[userInput()];  // Non-compliant
}
```

设 userInput 返回用户输入的整数，将其直接作为数组下标是不安全的。

应改为：

```
const char* bar() {
    int i = userInput();
    if (i >= 0 && i < 3) {
        return fruits[i];    // Compliant
    }
    return nullptr;
}
```

相关

ID_bufferOverflow

参考

CWE-119
CWE-125
CWE-131
CWE-787
CWE-788
C++ Core Guidelines ES.103
SEI CERT ARR30-C

R13.3 为缓冲区分配足够的空间

ID_insufficientBuffer　　buffer warning

为缓冲区分配足够的空间，避免溢出等问题。

示例：

```
void foo() {
    int* p = (int*)malloc(123);  // Non-compliant
    ....
}
```

例中 foo 函数为 int 型数组分配了 123 个字节的空间，而 123 不能被 sizeof(int) 整除，最后一个元素会越界。

应改为：

```
void foo() {
    int* p = (int*)malloc(123 * sizeof(int));  // Compliant
    ....
}
```

需要注意数组的逻辑大小和字节大小的区别，不可漏掉 sizeof 因子。

又如：

```
void bar(const char* s) {
    char* p = (char*)malloc(strlen(s));  // May be 'strlen(s) + 1'?
    ....
}
```

字符串以空字符结尾，在分配字符串空间时不可漏掉空字符的空间。

相关

ID_bufferOverflow

参考

CWE-131
CWE-135

R13.4 memset 等函数不应作用于带有虚函数的对象

ID_nonPODFilling　　buffer error

---

memset、memcpy、memmove 等具有填充功能的函数不应作用于带有虚函数的对象，否则会破坏其虚函数表等结构。

示例：

```
class A {
    int i;

public:
    virtual ~A();
};

void foo(A& a) {
    memset(&a, 0, sizeof(a));   // Non-compliant, the vftable is corrupted
}

void bar(A& a, A& b) {
    memcpy(&a, &b, sizeof(a));  // Non-compliant, the vftable is corrupted
}
```

参考

CWE-463
C++ Core Guidelines SL.con.4
C++ Core Guidelines C.90

R13.5 memset 等函数长度相关的参数不应有误

ID_badLength　　buffer error

对于 memset、memcpy、memmove、memcmp 及同类函数，表示长度的参数不应存在常见笔误。

示例：

```
char buf[1024];
memset(buf, 1024, 0);  // Non-compliant
```

应改为：

```
memset(buf, 0, 1024);  // Compliant
```

长度和填充值参数被写反是常见的笔误。

又如：

```
int arr[1024];
memset(buf, 0, 1024);  // Rather suspicious
memset(buf, 1, 123);   // Non-compliant
```

memset 等函数的长度单位为字节，不应与对象序列的逻辑长度有冲突，应改为：

```
memset(buf, 0, 1024 * sizeof(int));  // Compliant
memset(buf, 1, 123 * sizeof(int));   // Compliant
```

要注意不应遗漏 sizeof 因子。

又如，设 p 为对象的指针：

```
memset(p, 0, sizeof(p));   // Non-compliant
```

应改为：

```
memset(p, 0, sizeof(*p));  // Compliant
```

sizeof 作用于指针并不能获取到对象的大小，可参见 ID_sizeof_pointer 的进一步讨论。

又如，设 a、b 是对象：

```
memset(&a, 0, sizeof(&a));   // Non-compliant
memcpy(&a, &b, sizeof(&a));  // Non-compliant
```

应改为：

```
memset(&a, 0, sizeof(a));    // Compliant
memcpy(&a, &b, sizeof(a));   // Compliant
```

这是常见的复制粘贴错误。

又如：

```
if (memcmp(&a, &b, sizeof(a) != 0)) {  // Non-compliant
    ....
}
```

长度参数不应为比较表达式，应改为：

```
if (memcmp(&a, &b, sizeof(a)) != 0) {  // Compliant
    ....
}
```

括号的错误嵌套也是常见的笔误。

参考

CWE-130
CWE-805

R13.6 memset 等函数填充值相关的参数不应有误

ID_valueOverflow　　buffer error

---

memset、memset_s 等函数的填充值参数会被转为 unsigned char 型，所以其值不应超出一个字节的范围。

示例：

```
char buf[32];
memset(buf, 1024, 32);  // Non-compliant
```

例中填充值为 1024，超出了一个字节的范围，在实际代码中也可能是长度参数与填充值参数被写反了。

依据

ISO/IEC 9899:2011 7.24.6.1(2)
ISO/IEC 9899:2011 K.3.7.4.1(4)

参考

CWE-130

14. Pointer

R14.1 避免空指针解引用

ID_nullDerefInScp　　pointer error

解引用空指针会使程序产生崩溃等标准未定义的行为。

示例：

```
int foo(int i) {
    int* p = NULL;
    if (cond) {
        p = &i;
    }
    return *p;     // Non-compliant
}
```

例中指针 p 为空的状态可以到达解引用处，应避免这种问题。

```
void bar(T* p) {
    if (p) {
        p->baz();  // Compliant
    }
    p->qux();      // Non-compliant
}
```

例中对 qux 的调用超出了 p 的检查范围，这也是一种常见错误，应避免。

解引用空指针一般会使进程崩溃，给用户不好的体验，而且要注意如果崩溃可由外部输入引起，会被攻击者利用从而迫使程序无法正常工作，具有高可靠性要求的服务类程序更应该注意这一点，可参见“拒绝服务攻击”的进一步说明。对于客户端程序，也要防止攻击者对崩溃产生的“core dump”进行恶意调试，避免泄露敏感数据，总之程序的健壮性与安全性是紧密相关的。

依据

ISO/IEC 9899:1999 6.5.3.2(4)-undefined
ISO/IEC 9899:2011 6.5.3.2(4)-undefined

参考

CWE-476
C++ Core Guidelines ES.65

R14.2 注意逻辑表达式内的空指针解引用

ID_nullDerefInExp　　pointer error

在逻辑表达式中，需注意逻辑关系及运算符优先级，不可出现空指针解引用的问题。

示例：

```
bool fun0(T* p) {
    return p || p->foo();  // Non-compliant
}
```

当 p 为空时执行右子表达式，恰好产生空指针解引用问题。

```
bool fun1(T* p) {
    return p && p->foo() || p->bar();  // Non-compliant
}
```

由左子表达式可知 p 可能为空，而右子表达式却没有限制，有可能产生空指针解引用问题。

```
bool fun2(T* p) {
    return p->foo() && p;  // Non-compliant
}
```

这是颠倒了对 p 的判断和解引用次序，属于语言运用错误。

空指针解引用会导致标准未定义的错误，进程一般会崩溃，给用户不好的体验，而且要注意如果崩溃可由外部输入引起，会被攻击者利用从而迫使程序无法正常工作。

依据

ISO/IEC 9899:1999 6.5.3.2(4)-undefined
ISO/IEC 9899:2011 6.5.3.2(4)-undefined

参考

CWE-476
CWE-783
C++ Core Guidelines ES.65

R14.3 不可解引用已被释放的指针

ID_danglingDeref　　pointer error

已被释放的指针指向无效的内存空间，再次对其解引用会造成标准未定义的错误。

示例：

```
void foo() {
    int* p = new int[100];
    if (cond) {
        ....
        delete[] p;
    }
    do_somthing(p[0]);  // Non-compliant, ‘p’ may be deallocated
}
```

本来指针 p 指向有效的内存空间，但由于某种原因相关内存被释放，p 的值不变但已无效，这种情况被形象地称为“指针悬挂”， 未经初始化的指针和这种“被悬挂”的指针统称“野指针”，均指向无效地址不可被解引用。

在 C++ 语言中，应避免持有可自动销毁的对象地址，如容器中对象的地址、智能指针所指对象的地址等。

```
int bar(vector<int>& v) {
    int* p = &v.front();         // Bad practice
    v.push_back(some_value);
    return *p;                   // ‘p’ may be invalid
}
```

例中指针 p 记录了 vector 容器中对象的地址，根据 vector 容器持有对象的策略，随着元素的增加原有对象的地址可能不再有效。

又如：

```
int baz() {
    auto u = make_unique<int>();
    auto p = u.get();            // Bad practice
    u = make_unique<int>();
    return *p;                   // ‘p’ is invalid
}
```

例中指针 p 记录了 unique_ptr 所指对象的地址，当 unique_ptr 指向新的对象时，原对象的地址不再有效。

相关

ID_illAccess

依据

ISO/IEC 9899:1999 6.5.3.2(4)-undefined
ISO/IEC 9899:2011 6.5.3.2(4)-undefined

参考

CWE-822
CWE-825
C++ Core Guidelines ES.65

R14.4 避免无效的空指针检查

ID_invalidNullCheck　　pointer warning

当指针的值一定不为空时，再对其进行检查是没有意义的，往往意味着逻辑错误。

示例：

```
if (int* p = new int[100]) {  // Non-compliant
    ....
} else {  // Invalid
    ....
}
```

标准规定默认 new 运算符的返回值不会为空，如果分配失败则抛出异常，所以这种检查和相关错误处理是无效的。

应改为：

```
if (int* p = new(std::nothrow) int[100]) {  // Compliant
    ....
} else {  // OK
    ....
}
```

又如：

```
if (p) {  // Meaningless
    delete p;
}
```

对于可接受空指针的接口，不必总在调用前判断指针是否为空，否则会使代码变得繁琐。delete 关键字或 free 函数可以作用于空指针，调用之前的检查是没有意义的。

相关

ID_repeatedNullCheck

依据

ISO/IEC 9899:2011 18.6

R14.5 不应重复检查指针是否为空

ID_repeatedNullCheck　　pointer warning

重复的空指针检查是不必要的，使代码显得繁琐，且干扰编译器优化。

示例：

```
void foo(int* p) {
    if (!p) {
        return;
    }
    if (p) {      // Non-compliant, p is not nullptr
        ....
    } else {
        ....      // Unreachable
    }
}
```

相关

ID_invalidNullCheck

R14.6 不应将非零常量值赋值给指针

ID_fixedAddrToPointer　　pointer warning

固定地址是不可移植的，且存在安全隐患。

示例：

```
const void* badAddr = (void*)0xffffffff;  // Non-compliant
```

示例代码的本意是声明一个表示无效地址的值，但在 64 位系统中这个地址可能是有效的。

又如：

```
typedef int (*fp_t)(int);
fp_t fp = (fp_t)0x1234abcd;  // Non-compliant
int res = (*fp)(123);        // Unsafe
```

示例代码假设在特定地址可以找到特定的函数，将该地址赋给一个指针并调用，这种假设本身可能就是错误的，会导致崩溃，攻击者也可以更改预期地址上的内存，从而导致任意代码的执行。

某些框架或系统会以 -1 表示无效地址，但不具备通用性，不妨通过配置选择是否放过。

配置

allowMinusOneAsPointerValue：为 true 时可以放过 -1 作为指针值的情况

相关

ID_ptrIntCast

参考

CWE-587

R14.7 不应使用 bool 常量对指针赋值或初始化

ID_oddPtrBoolAssignment　　pointer warning

用 false 等 bool 常量对指针赋值或初始化是非常怪异的，会对代码阅读造成误导，而且也可能是书写错误。

示例：

```
void set_false(bool* p) {
    p = false;  // Non-compliant, should be ‘*p = false’
}
```

参考

CWE-351

R14.8 不应使用字符常量对指针赋值或初始化

ID_oddPtrCharAssignment　　pointer warning

用 '\0' 、L'\0' 等字符常量对指针赋值或初始化是非常怪异的，往往意味错误。

示例：

```
void set_terminate(char* p) {
    p = '\0';  // Non-compliant
}
```

应改为：

```
void set_terminate(char* p) {
    *p = '\0';  // Compliant
}
```

参考

CWE-351

R14.9 不应使用常数 0 对指针赋值

ID_zeroAsPtrValue　　pointer suggestion

不应使用常数 0 对指针赋值，在 C++ 代码中应使用 nullptr，在 C 代码中应使用 NULL，否则易出现类型转换相关的错误，且不利于阅读。

示例：

```
int foo(int* p);

int* p = 0;      // Non-compliant
int i = foo(0);  // Non-compliant
```

应改为：

```
int* p = nullptr;      // Compliant
int i = foo(nullptr);  // Compliant
```

参考

MISRA C++ 2008 4-10-2
C++ Core Guidelines ES.47

R14.10 指针不应与 bool 常量比较大小

ID_oddPtrBoolComparison　　pointer warning

指针与 bool 常量比较大小是非常怪异的，往往是某种笔误。

示例（设 p 为指针）：

```
p == false  // Non-compliant
p != false  // Non-compliant
```

如果判断指针是否为空，只应将指针与 NULL 或 nullptr 比较，其他常量均不合规。在 C++ 语言中应优先使用 nullptr。

参考

CWE-1025

R14.11 指针不应与字符常量比较大小

ID_oddPtrCharComparison　　pointer warning

指针与字符常量比较大小是非常怪异的，往往是某种笔误。

示例（设 p 为指针）：

```
p == '\0'    // Non-compliant
p == L'\0'   // Non-compliant
```

这种情况很有可能是漏写了 * 号：

```
*p == '\0'   // Compliant
*p == L'\0'  // Compliant
```

否则只应将指针与 NULL 或 nullptr 比较。

参考

CWE-1025

R14.12 不应判断指针大于、大于等于、小于、小于等于 0

ID_oddPtrZeroComparison　　pointer warning

指针的值是地址的编号，并没有大小的语义，指针可以与指针比较从而确定某种前后关系，但指针与整数比较大小则是没有意义的，尤其是与 0 的比较，往往意味着错误。

示例（设 p 为指针）：

```
p < 0    // Non-compliant, always false
p >= 0   // Non-compliant, always true
p > 0    // Non-compliant, use p != nullptr instead
p <= 0   // Non-compliant, use p == nullptr instead
```

另外，指针与 NULL 或 nullptr 只应使用 == 或 != 进行比较，其他比较运算符均不合规：

```
p <= NULL      // Non-compliant
p > nullptr    // Non-compliant
```

应改为：

```
p == NULL      // Compliant
p != nullptr   // Compliant
```

参考

CWE-1025

R14.13 不应判断 this 指针是否为空

ID_this_zeroComparison　　pointer warning

正常情况下 this 指针不会为空，而且判断 this 指针是否为空会影响编译器对 this 指针的优化，造成难以预料的后果。

在某些环境中用空指针调用非静态成员函数时，this 指针可能为空，但这并不符合标准。值得强调的是，任何情况下都不应逃避解引用空指针造成的问题。

示例：

```
struct A {
    int x = 0;
    int getX() {
        return this? x: 0;  // Non-compliant
    }
};

A* p = foo();
// Suppose an error has occurred and a null pointer is returned
cout << p->getX() << '\n';
```

假设 foo 函数不应返回空指针，而某个错误导致其返回了空指针，程序本应崩溃，而 getX 函数却逃避了崩溃，这非但不能真正地解决问题，反而使问题难以定位，使程序难以调试，大大降低了可维护性。

依据

ISO/IEC 14882:2003 4.1(1)
ISO/IEC 14882:2011 4.1(1)

参考

CWE-1025

R14.14 析构函数中不可使用 delete this

ID_this_deleteInDestructor　　pointer error

析构函数中不可使用 delete this，否则造成无限递归。

示例：

```
struct A {
    ~A() {
        delete this;  // Non-compliant
    }
};
```

参考

CWE-674

R14.15 禁用 delete this

ID_this_forbidDeleteThis ⛔ pointer suggestion

因为正确使用 delete this 限制条件太多，稍不留意就会为 bug 埋下伏笔，所以禁用这种方式是明智的选择。

要想正确使用 delete this，必须保证：

- 对象是用 new 创建的，但不能是 new[] 或 replacement new
- 使用 delete this 之后不能再访问相关非静态成员及成员函数
- 不能在析构函数中使用 delete this

示例：

```
class A {
    ....
public:
    void foo() {
        delete this;  // Non-compliant
    }
};

auto* p = new A;
p->foo();         // Looks innocent

p = new A[10];
p->foo();         // Memory is still leaking
```

如果一定要使用 delete this，类的析构函数应设为私有，这样可以阻止类的对象在栈上定义而引发更大的混乱，并且要确保执行 delete this 后 this 指针再也不会被解引用，而且不能用 new[] 创建，否则仍然存在内存泄漏问题。

所以对框架以及语言工具之外的业务类或算法类代码建议禁用 delete this。

R14.16 sizeof 作用于指针是可疑的

ID_sizeof_pointer ❓ pointer suspicious

---

sizeof 作用于指针获取到的是指针变量的大小，而不是指针指向内容的大小，sizeof 作用于指针很容易造成错误。

示例：

```
void foo(int* p) {
    memset(p, 0, sizeof(p));   // Logic error
}
```

应改为：

```
void foo(int* p, int n) {
    memset(p, 0, n * sizeof(*p));  // OK
}
```

其中参数 n 是数组元素的个数。

相关

ID_sizeof_pointerDivision

参考

CWE-467

R14.17 判断 dynamic_cast 转换是否成功

ID_nullDerefDynamicCast　　pointer warning

---

dynamic_cast 转换指针失败会返回空指针，转换引用失败会抛出异常，如果不作判断则失去了使用 dynamic_cast 的意义。

示例：

```
void foo(A* a) {
    dynamic_cast<B*>(a)->foo();  // Non-compliant
}
```

应改为：

```
void foo(A* a) {
    if (auto* b = dynamic_cast<B*>(a)) {  // Compliant
        b->foo();
    }
}
```

使用 dynamic_cast 需要一定开销，如果不对其结果作判断，还不如使用 static_cast 等转换，但本规则集合不建议采用非 dynamic_cast 之外的向下类型转换，可参见 ID_nonDynamicDownCast 的相关讨论。

相关

ID_nonDynamicDownCast

依据

ISO/IEC 14882:2011 5.2.7(9)

参考

CWE-476
C++ Core Guidelines C.148

R14.18 指针在释放后应置空

ID_missingResetNull　　pointer suggestion

指针指向的动态内存空间被回收后指针不再有效，这时应将指针设为空指针，可避免重复释放造成的问题，如果后续对指针仍有错误的读写，也可使问题立即显现出来，不至于造成难以排查的问题。

尤其是在析构函数中，建议所有需要释放的成员指针在释放后都置为空指针。

示例：

```
class T {
    int* p = new int[123];

public:
   ~T() {
        delete[] p;
        p = nullptr;  // Good
    }
};
```

例中指针 p 被释放之后置为 nullptr，如果 T 的析构函数被外界反复调用也不会造成问题。本规则是关于“指针悬挂”问题的有效措施，可参见 ID_danglingDeref 的进一步讨论。

相关

ID_danglingDeref
ID_explicitDtorCall

参考

SEI CERT MEM01-C

15. Style

R15.1 空格应遵循统一风格

ID_spaceStyle　　style suggestion

与运算符、标点符、关键字相关的空格应遵循统一风格，过于随意的空格会干扰阅读，甚至形成笔误。

本规则暂不限定具体风格，但强调一致性，同类运算符、标点符、关键字的空格方式应保持一致，tab 等变长空白符不应用作空格。

示例：

```
a= 0;                   // Bad
b = a +1;               // Bad
c =~ a;                 // Bad
while (d --> 0)         // Bad

a == 0 ? foo() :bar();  // Bad
baz (1 , 2 ,3);         // Bad

if (cond)
if(cond)                // Missing a unified style
```

例中运算符和关键字相关的空格风格不一致，代码显得很混乱。

本规则是 ID_stickyAssignmentOperator 的泛化，该规则描述了一种由空格造成的错误。

相关

ID_stickyAssignmentOperator

R15.2 大括号应遵循统一风格

ID_braceStyle　　style suggestion

大括号应遵循统一的换行和缩进风格，否则会干扰阅读，甚至形成笔误。

命名空间、类、函数体、复合语句等不同类别的大括号，换行方式可以不同，但同类大括号的换行方式应该是一致的，本规则暂不限定具体风格，但强调一致性。

示例：

```
void foo() {
    if (cond)
    {
        ....
    }
    else {
        ....
    }
}

void bar()
{
    if (cond) {
        ....
    } else
    {
        ....
    }
}
```

例中大括号换行的方式不一致，代码显得很混乱。

本规则是 ID_if_mayBeElseIf 的泛化，该规则描述了一种由换行造成的错误。

相关

ID_if_mayBeElseIf

R15.3 NULL 和 nullptr 不应混用

ID_mixNullptrAndNULL　　style warning

NULL 和 nullptr 不应混用，应统一使用 nullptr。

示例：

```
void foo(int* a = NULL, int* b = nullptr);     // Non-compliant
void bar(int* a = nullptr, int* b = nullptr);  // Compliant
```

相关

ID_deprecatedNULL

参考

C++ Core Guidelines ES.47

R15.4 在 C++ 代码中用 nullptr 代替 NULL

ID_deprecatedNULL　　style suggestion

在 C++ 语言中，标识符 NULL 虽然可以用来表示空指针，但该标识符是由实现定义的，而且往往不能有效区分整型常量 0 和空指针，根据 C++11 标准，应使用 nullptr 表示空指针。

示例：

```
void foo(int*) {
    cout << "foo(int*)\n";
}

void foo(int) {
    cout << "foo(int)\n";
}

int main() {
    foo(NULL);  // Non-compliant, what is output?
}
```

显然，例中 main 函数中的 foo 函数预期应调用参数为指针的版本，然而不同的编译器对这段代码有不同的行为，有的无法通过编译，有的编译执行后会输出“foo(int)”，用 nullptr 代替 NULL 可解决这种问题。

依据

ISO/IEC 9899:1999 7.17(3)-implementation
ISO/IEC 9899:2011 7.19(3)-implementation
ISO/IEC 14882:2003 C.2.2.3(1)-implementation
ISO/IEC 14882:2011 C.3.2.4(1)-implementation
ISO/IEC 14882:2017 C.5.2.7(1)-implementation
ISO/IEC 14882:2011 2.14.7(1)

参考

C++ Core Guidelines ES.47

R15.5 赋值表达式不应作为子表达式

ID_assignmentAsSubExpression　　style suggestion

赋值表达式作为子表达式增加了复杂性，且容易产生优先级相关的问题。

普通赋值表达式、复合赋值表达式均受本规则约束。

示例：

```
int a, b = foo();
if (a = b != 0) {    // Non-compliant
    ....
}
int c = bar();
return a += b += c;  // Non-compliant
```

参考

CWE-481
MISRA C 2004 13.1
MISRA C 2012 13.4
MISRA C++ 2008 6-2-1

R15.6 不应存在多余的分号

ID_redundantSemicolon　　style suggestion

多余的分号使代码显得繁琐，也可能包含某种错误，应该去掉。

示例：

```
namespace N {
    void foo() {
        bar();;    // Non-compliant
    };             // Non-compliant
};                 // Non-compliant
```

结语

保障软件安全、提升产品质量是宏大的主题，需要不断地学习、探索与实践，也难以在一篇文章中涵盖所有要点，这 417 条规则就暂且讨论至此了。欢迎提供修订意见和扩展建议，由于本文档是自动生成的，请不要直接编辑本文档，可在 Issue 区发表高见，管理员修正数据库后会在致谢列表中存档。

此致

祝编程愉快！