ZTE中兴

内部公开▲

技术标准

Q/ZX

中 兴 通 讯 股 份 有 限 公 司 企 业 标 准

Q/ZX 00114.33-2017

C++安全编码规范

2017-6-30 发布　　2017-7-1 实施

中兴通讯股份有限公司 发　布

前　　言

为了提高公司软件产品的安全属性，公司对安全编码进行统一的规范约束。

本规范是安全编码系列规范的第 2 部分，整个安全编码规范包括：

第 1 部分 C 语言安全编码规范

第 2 部分 C++安全编码规范

第 3 部分 Java 安全编码规范

本标准由中兴通讯产品安全部提出，技术规划部、产品安全部归口管理。

本标准于 2017 年 6 月首次发布。

为便于理解，保留最近三次修订记录。

标准编号	修改前编号	主要修订内容	修订部门	修订人	审核人	批准人
Q/ZX 00114.33-2017		首次发布	LTE 开发二部， 多业务承载深圳软件开发一部， 虚拟化西安开发部， BBU 基带部， LTE 开发二部， MANO 一部， 技术规划部	李永顺 10110636， 李道春 10073354， 张晓龙 10065132， 李小海 10072984， 邓传斌 10098481， 曾亮亮 10091330， 傅锦华 10108953， 刘光聪 10209986	王华刚 10059176	钟宏 10000263

C++ 安全编码规范

TIPS: 请打开 PDF 工具的书签栏目。

- Rule 01. 声明及初始化
  - DCL50-CPP. 不要定义C风格的变参函数
  - DCL51-CPP. 不要声明或定义保留的标识符
  - DCL52-CPP. 不要使用const或volatile限定引用变量
  - DCL53-CPP. 语句声明时不要有语法歧义
  - DCL54-CPP. 成对重载allocation与deallocation函数并置于相同作用域中
  - DCL55-CPP. 避免跨信任边界传递类对象时发生信息泄露
  - DCL56-CPP. 避免循环初始化静态对象
  - DCL57-CPP.不要让异常逃出析构函数或内存释放函数
  - DCL58-CPP. 不要修改标准命名空间
  - DCL59-CPP. 不要在头文件中定义匿名的命名空间
- Rule 02. 表达式
  - EXP50-CPP. 不要依赖求值顺序
  - EXP51-CPP. 不要通过不正确类型指针删除一个数组
  - EXP52-CPP. 不要依赖不求值的操作数
  - EXP53-CPP. 不要读取未初始化的内存
  - EXP54-CPP. 不要访问在其生命周期之外的对象
  - EXP55-CPP. 不要通过cv-unqualified类型访问一个cv-qualified对象
  - EXP57-CPP. 不要强制转换或删除指向不完整类的指针
  - EXP58-CPP. 给va_start传递正确类型的对象
  - EXP61-CPP. lambda对象不能超出其捕获引用对象的生命周期
  - EXP62-CPP. 不要访问对象中非对象值部分的bits
  - EXP63-CPP. 不要依赖移动对象的值
- Rule 03. 整型
  - INT50-CPP. 不要做超过枚举值范围的枚举类型转换
- Rule 04. 容器
  - CTR50-CPP. 保证容器的下标和迭代器在有效的范围内(刘涛)
  - CTR51-CPP. 用有效的引用、指针或者迭代器来获取容器中的元素(刘涛)
  - CTR52-CPP. 保证库函数不会溢出
  - CTR53-CPP. 使用有效的迭代器范围
  - CTR54-CPP. 不要减去来自不同容器的迭代器
  - CTR55-CPP. 在一个可能导致溢出的迭代器上不要使用添加操作符
  - CTR56-CPP. 不要在一个多态对象上进行指针运算
  - CTR57-CPP. 提供有效的排序谓词

  - CTR58-CPP. 谓词函数对象不应该可变

- Rule 05. 字符和字符串

  - STR50-CPP. 保证字符串有足够的存储空间存放字符数据和结束符标识
  - STR51-CPP. 不要试图用空指针创建std::string
  - STR52-CPP. 使用有效的引用、指针和迭代器来引用basic_string中的元素
  - STR53-CPP. 元素访问范围检查

- Rule 06. 内存管理

  - MEM50-CPP. 禁止访问已释放的内存
  - MEM51-CPP. 正确释放动态分配的资源
  - MEM53-CPP. 手动管理对象的生命周期时，显式的构造和析构对象
  - MEM55-CPP. 谨慎替换动态内存管理
  - MEM56-CPP. 不要使用不关联的智能指针存储一个已被接管的指针值

- Rule 07. 输入与输出

  - FIO50-CPP. 不要在没有定位操作的情况下交替地从同一个文件流中输入输出
  - FIO51-CPP. 不再需要的时候就关闭文件

- Rule 08. 异常与错误处理

  - ERR50-CPP. 不要突然终止程序
  - ERR51-CPP. 处理所有异常
  - ERR52-CPP. 不要使用setjmp()或longjmp()
  - ERR53-CPP. 不要在构造或析构方法的function-try-block句柄中引用基类或类的成员变量
  - ERR54-CPP. 异常的捕获句柄应该按参数类型来排序，从最深的派生到最浅的派生
  - ERR55-CPP. 规范异常说明
  - ERR56-CPP. 保证异常能被安全地处理
  - ERR57-CPP. 不要在处理异常时泄露资源
  - ERR58-CPP. 在main方法开始处理时处理所有的异常抛出
  - ERR59-CPP. 不要在执行边界上抛出异常
  - ERR60-CPP. 异常对象必须是无异常抛出的拷贝构造
  - ERR61-CPP. 通过左值引用来捕获异常

- Rule 09. 面向对象编程

  - OOP50-CPP. 禁止在构造函数或析构函数中调用虚函数
  - OOP51-CPP. 禁止切割子类对象
  - OOP52-CPP. 禁止删除没有虚拟析构函数的多态对象
  - OOP53-CPP. 规范化成员初始化列表的顺序
  - OOP54-CPP. 优雅地处理“自我赋值”

  - OOP55-CPP. 禁止使用指向成员的指针访问不存在的成员
  - OOP56-CPP. 遵循替换处理程序的规则
  - OOP57-CPP. 特殊成员函数和重载运算符优于C标准库函数
  - OOP58-CPP. 拷贝操作不允许修改原对象

- Rule 10. 并发

  - CON50-CPP. 不要销毁一个正在被锁定的互斥对象
  - CON51-CPP. 确保异常条件下持有的锁被释放
  - CON52-CPP. 多个线程访问位字段时需要防止数据竞争
  - CON53-CPP. 按预定义的顺序加锁来避免死锁
  - CON54-CPP. 在循环中封装那些可能产生伪激活的函数
  - CON55-CPP. 使用条件变量时保持线程的安全性和活性
  - CON56-CPP. 不要刻意地锁一个已经被其它线程持有的非递归锁

- Rule 11. 杂项

  - MSC50-CPP. 禁止使用 std::rand() 生成伪随机数
  - MSC51-CPP. 确保随机数生成引擎使用合适的种子
  - MSC52-CPP. 带返回值的函数必须在每个路径中返回返回值
  - MSC54-CPP. 信号处理程序必须是POF

Rule 01. 声明及初始化

DCL50-CPP. 不要定义C风格的变参函数

作者：罗胜金10041900 评审人： 刘光聪10209986

C++提供两种机制定义变参函数：一为函数参数包，二为在函数声明时使用C风格的省略号作为参数。
本规则要求：不要定义C风格的省略号作为函数参数，因为它导致无法检查参数类型、无法表达参数语义。

不合规代码

以下代码连续读入多个整数并求和，从第3个参数开始若遇到0结束。如果入参没有0，或者入参为非整数，将导致不可预知结果。

```
#include <cstdarg>
int add(int first, int second, ...) {
```

```
  int r = first + second;
  va_list va;
  va_start(va, second);
  while (int v = va_arg(va, int)) {
    r += v;
  }
  va_end(va);
  return r;
}
```

合规代码（递归包扩展）

以下代码使用函数参数包方式，实现变参的 `add()` 函数。如果入参没有0，不会导致不可预知结果。另外，如果入参为非整数，则 `std::enable_if` 确保函数是病态（ill-formed）的，也不会导致不可预知结果。

```
#include <type_traits>

template <typename Arg, typename std::enable_if<std::is_integral<Arg>::value>::type
* = nullptr>
int add(Arg f, Arg s) { return f + s; }

template <typename Arg, typename... Ts, typename std::enable_if<std::is_integral<Arg
>::value>::type * = nullptr>
int add(Arg f, Ts... rest) {
  return f + add(rest...);
}
```

合规代码（初始化列表扩展）

本例将函数参数包扩展为一个值列表，并作为初始化列表的一部分。由于初始化列表中不允许收缩转换（narrowing conversions），所以这里的 `std::enable_if` 不涉及变参保护，仅保证类型安全。

```
#include <type_traits>

template <typename Arg, typename... Ts, typename std::enable_if<std::is_integral<Arg
>::value>::type * = nullptr>
int add(Arg i, Arg j, Ts... all) {
  int values[] = { j, all... };
  int r = i;
```

```
  for (auto v : values) {
    r += v;
  }
  return r;
}
```

参考文献

- DCL50-CPP

DCL51-CPP. 不要声明或定义保留的标识符

作者：罗胜金10041900 评审人： 刘光聪10209986

禁止在代码中声明或定义C++的保留标识符，包括但不限于 `override, final, alignas, carry_dependency, deprecated, noreturn` 等。声明或定义保留标识符，将导致不可预知结果。

不合规代码（头文件保护）

以下代码中的头文件保护宏，可能与C++标准库中的保留名称冲突。即使未包含C++标准库，也可能与编译器隐式预定义的保留名称冲突。原因是C++保留名称通常以 _ 或 __ 开头。

```
#ifndef _MY_HEADER_H_
#define _MY_HEADER_H_

// Contents of <my_header.h>

#endif // _MY_HEADER_H_
```

合规代码（头文件保护）

不在头文件保护宏中使用前缀或者后缀 _ ，避免与C++标准库或编译器保留名称冲突。

```
#ifndef MY_HEADER_H
#define MY_HEADER_H

// Contents of <my_header.h>
```

```
#endif // MY_HEADER_H
```

不合规代码（用户定义操作符）

以下代码中，用户自定义 operator "" x 。由于C++保留无下划线前缀的运算符，用于将来库的实现，因此，用户自定义运算符应该以下划线 _ 为前缀。

```
#include <cstddef>

unsigned int operator"" x(const char *, std::size_t);
```

合规代码（用户定义操作符）

以下代码中，用户自定义 operator ""_x ，它不是保留标识符。

```
#include <cstddef>

unsigned int operator"" _x(const char *, std::size_t);
```

说明：用户自定义运算符是 operator ""_x ，而不是 _x ， _x 为全局命名空间保留。

不合规代码（文件作用域）

下面代码中， _max_limit 和 _limit 的名称都以下划线开头。 _max_limit 被限定在本文件中使用，可能与 <cstddef> 中的符号发生名称冲突。 _limit 可能与C++运行库产生名称冲突，即使该符号未在任何头中声明。总之，定义任何下划线开头的全局符号名称都是不安全的。

```
#include <cstddef> // std::for size_t

static const std::size_t _max_limit = 1024;
std::size_t _limit = 100;

unsigned int get_value(unsigned int count) {
  return count < _limit ? count : _limit;
}
```

合规代码（文件作用域）

下面代码中，全局符号名称没有以 _ 开头。

```
#include <cstddef> // for size_t

static const std::size_t max_limit = 1024;
std::size_t limit = 100;

unsigned int get_value(unsigned int count) {
  return count < limit ? count : limit;
}
```

不合规代码（保留宏）

在下面代码中，名称 `MAX_SIZE` ，与 `<cstdint>` 中用于表示 `std::size_t` 上限的宏名称冲突。

```
#include <cinttypes> // for int_fast16_t

void f(std::int_fast16_t val) {
  enum { MAX_SIZE = 80 };
  // ...
}
```

合规代码（模块前缀宏）

在下面代码中，为避免重定义保留名字，增加用户模块名作为前缀。

```
#include <cinttypes> // for std::int_fast16_t

void f(std::int_fast16_t val) {
  enum { MYMODULE_MAX_SIZE = 80 };
  // ...
}
```

参考文献

DCL51-CPP

DCL52-CPP. 不要使用const或volatile限定引用变量

作者：罗胜金10041900 评审人： 刘光聪10209986

C++禁止或忽略cv（const或volatile）标识符对引用类型变量的限定。
例如，可以使用 `char const &p;` 或者 `const char &p;` ，限定引用目标是不可修改的。但不要试图使用 `char &const p;` 来限定引用类型变量不可修改。如果试图用cv限定引用类型变量，可能导致不可预知结果。

不合规代码

如下代码，使用const引用限定 `char` ，代替一个const `char` 的引用。这可能导致不可预知结果。

```
#include <iostream>

void f(char c) {
  char &const p = c;
  p = 'p';
  std::cout << c << std::endl;
}
```

使用Microsoft Visual Studio 2015，上面代码编译成功，并带有如下警告：

> *warning C4227: anachronism used : qualifiers on reference are ignored .*
> *运行时，代码输出: p 。*
> *使用Clang 3.9，此代码编译失败，错误信息如下：*
> *error: 'const' qualifier may not be applied to a reference*

不合规代码

如下代码中，虽然正确的声明了 `p` ，但随后对 `p` 的修改使程序出现错误：

```
#include <iostream>

void f(char c) {
  const char &p = c;
  p = 'p'; // Error: read-only variable is not assignable
  std::cout << c << std::endl;
}
```

合规代码

下面代码删除了const限定符：

```
#include <iostream>

void f(char c) {
  char &p = c;
  p = 'p';
  std::cout << c << std::endl;
}
```

参考文献

DCL52-CPP

DCL53-CPP. 语句声明时不要有语法歧义

作者：罗胜金10041900 评审人： 刘光聪10209986

某些代码语句语义模糊，编译器无法确定它是表达式还是声明，或者无法确定它是函数声明还是变量初始化。要避免出现这些情况。

不合规代码

如下代码，既可以解释为声明一个 `std::unique_lock` 类型的匿名对象，并调用其单参数的构造函数，又可以被解释为声明一个名为 `m` 的对象，并通过默认构造函数进行构造。此例被解释为后者。

```
#include <mutex>

static std::mutex m;
static int shared_resource;

void increment_by_42() {
  std::unique_lock<std::mutex>(m);
  shared_resource += 42;
}
```

合规代码

下面代码中，明确锁对象名称为lock，并调用参数为m的构造函数，避免了歧义。

```
#include <mutex>

static std::mutex m;
static int shared_resource;

void increment_by_42() {
  std::unique_lock<std::mutex> lock(m);
  shared_resource += 42;
}
```

不合规代码

如下代码，f()函数语义不明确，既可以理解为声明一个没有参数、返回值为 `Widget` 的函数指针，又可以理解为声明 `Widget` 类型的局部变量。此例被解释为前者，因此 `Widget` 的默认构造函数未被调用。

```
#include <iostream>

struct Widget {
  Widget() { std::cout << "Constructed" << std::endl; }
};

void f() {
  Widget w();
}
```

合规代码

如下代码给出两种合规的声明，第一种是在变量声明之后删除括号，确保进行变量声明而不是函数声明。第二种是使用 `braced-init-list` 直接初始化局部变量。

```
#include <iostream>

struct Widget {
  Widget() { std::cout << "Constructed" << std::endl; }
};
```

```
void f() {
  Widget w1; // Elide the parentheses
  Widget w2{}; // Use direct initialization
}
```

不合规代码

编译器对如下代码的解析令人费解， `Gadget g(Widget(i));` 不会被解析为定义一个带初始化参数的 `Gadget` 对象，而是被解析为一个函数声明 `Gadget g(Widget i);` ，在参数 `i` 周围有一组冗余的括号。因此， `f()` 函数没有构建 `Gadget` 或 `Widget` 对象。

```
#include <iostream>

struct Widget {
  explicit Widget(int i) { std::cout << "Widget constructed" << std::endl; }
};

struct Gadget {
  explicit Gadget(Widget wid) { std::cout << "Gadget constructed" << std::endl; }
};

void f() {
  int i = 3;
  Gadget g(Widget(i));
  std::cout << i << std::endl;
}
```

合规代码

如下代码给出两种正确构造 `g` 的方式，第一个声明 `g1` 在构造函数调用的参数周围增加一组额外的括号，迫使编译器将其解析为 `Gadget` 类型的局部变量声明，而不是函数声明。 第二个声明 `g2` ，直接使用 `{}` 进行初始化。

```
#include <iostream>

struct Widget {
  explicit Widget(int i) { std::cout << "Widget constructed" << std::endl; }
};

struct Gadget {
```

```
  explicit Gadget(Widget wid) { std::cout << "Gadget constructed" << std::endl; }
};

void f() {
  int i = 3;
  Gadget g1((Widget(i))); // Use extra parentheses
  Gadget g2{Widget(i)}; // Use direct initialization
  std::cout << i << std::endl;
}
```

运行此程序将产生预期输出：

```
Widget constructed
Gadget constructed
Widget constructed
Gadget constructed
3
```

参考文献

DCL53-CPP

DCL54-CPP. 成对重载allocation与deallocation函数并置于相同作用域中

作者：罗胜金10041900 评审人： 刘光聪10209986

分配内存函数和释放内存函数，均可以在全局或类作用域内重载。要求分配内存和释放内存函数成对重载，并且作用域相同。

不合规代码

下面代码中，分配函数在全局范围重载，但是未声明相应的释放函数。如果使用重载的分配函数生成一个对象，删除对象时可能导致不可预知结果。

```
#include <Windows.h>
#include <new>

void *operator new(std::size_t size) noexcept(false) {
```

```
  static HANDLE h = ::HeapCreate(0, 0, 0); // Private, expandable heap.
  if (h) {
    return ::HeapAlloc(h, 0, size);
  }
  throw std::bad_alloc();
}

// No corresponding global delete operator defined.
```

合规代码

下面代码，在全局范围内定义相应的释放内存函数。

```
#include <Windows.h>
#include <new>

class HeapAllocator {
  static HANDLE h;
  static bool init;

public:
  static void *alloc(std::size_t size) noexcept(false) {
    if (!init) {
      h = ::HeapCreate(0, 0, 0); // Private, expandable heap.
      init = true;
    }

    if (h) {
      return ::HeapAlloc(h, 0, size);
    }
    throw std::bad_alloc();
  }

  static void dealloc(void *ptr) noexcept {
    if (h) {
      (void)::HeapFree(h, 0, ptr);
    }
  }
};

HANDLE HeapAllocator::h = nullptr;
bool HeapAllocator::init = false;
```

```
void *operator new(std::size_t size) noexcept(false) {
  return HeapAllocator::alloc(size);
}

void operator delete(void *ptr) noexcept {
  return HeapAllocator::dealloc(ptr);
}
```

不合规代码

如下代码中， operator new() 在类范围内重载，但 operator delete() 在类范围内没有相应重载。尽管重载的分配函数实际调用默认全局分配函数，但由于未能相应地更新分配地址，删除对象时将导致不可预知结果。

```
#include <new>

extern "C++" void update_bookkeeping(void *allocated_ptr, std::size_t size, bool all
oc);

struct S {
  void *operator new(std::size_t size) noexcept(false) {
    void *ptr = ::operator new(size);
    update_bookkeeping(ptr, size, true);
    return ptr;
  }
};
```

合规代码

如下代码中， operator delete() 相应地在类范围内被定义。

```
#include <new>

extern "C++" void update_bookkeeping(void *allocated_ptr, std::size_t size, bool all
oc);

struct S {
  void *operator new(std::size_t size) noexcept(false) {
    void *ptr = ::operator new(size);
    update_bookkeeping(ptr, size, true);
```

```
    return ptr;
  }

  void operator delete(void *ptr, std::size_t size) noexcept {
    ::operator delete(ptr);
    update_bookkeeping(ptr, size, false);
  }
};
```

参考文献

DCL54-CPP

DCL55-CPP. 避免跨信任边界传递类对象时发生信息泄露

作者：罗胜金10041900 评审人： 刘光聪10209986

C++结构体经常包含填充位，当数据在不同信任域的结构体之间拷贝时，如果源结构体的填充位包含敏感信息，则可能导致信息泄露。本条规则要求，程序员必须确保，在进行结构体拷贝时，不拷贝填充信息。

不合规代码

下面代码，结构体使用填充位确保成员对齐。当结构体数据复制到用户空间时，这些填充位包含的敏感信息可能泄露。

```
#include <cstddef>
struct test {
    int a;
    char b;
    int c;
};
// 安全复制字节到用户空间
extern int copy_to_user(void *dest, void *src, std::size_t size);
void do_stuff(void *usr_buf) {
    test arg{1, 2, 3};
    copy_to_user(usr_buf, &arg, sizeof(arg));
}
```

不合规代码

看起来，下面代码中，初始化列表会将对象中所有比特初始化为 `0` 。但是，编译器可能将32位寄存器的低字节设置为 `2` ，高字节不变，再存储整个32位数据，仍然存在信息泄露风险。

```
#include <cstddef>
struct test {
int a;
char b;
int c;
};
//安全复制字节到用户空间
extern int copy_to_user(void *dest, void *src, std::size_t size);
void do_stuff(void *usr_buf) {
test arg{};
    arg.a = 1;
    arg.b = 2;
    arg.c = 3;
copy_to_user(usr_buf, &arg, sizeof(arg));
}
```

合规代码

以下代码，先将结构数据序列化，再复制到不受信任的上下文。

```
#include <cstddef>
#include <cstring>
struct test {
    int a;
    char b;
    int c;
};
// 安全复制字节到用户空间
extern int copy_to_user(void *dest, void *src, std::size_t size);
void do_stuff(void *usr_buf) {
    test arg{1, 2, 3};
// 可能大于严格需要的内存大小
    unsigned char buf[sizeof(arg)];
    std::size_t offset = 0;
    std::memcpy(buf + offset, &arg.a, sizeof(arg.a));
    offset += sizeof(arg.a);
```

```
    std::memcpy(buf + offset, &arg.b, sizeof(arg.b));
    offset += sizeof(arg.b);
    std::memcpy(buf + offset, &arg.c, sizeof(arg.c));
    offset += sizeof(arg.c);
    copy_to_user(usr_buf, buf, offset /* size of info copied */);
}
```

这段代码保证了复制到非特权用户的填充位都是被初始化过的。复制到用户空间的结构现在是打包的，需要用 copy_to_user() 函数解包以重建原始的填充结构。

合规代码（填充字节）

以下代码，把填充位显式声明为结构中的字段。但是，这个解决方案不可移植，它特定用于x86-32体系结构。

```
#include <cstddef>
struct test {
    int a;
    char b;
    char padding_1, padding_2, padding_3;
    int c;
    test(int a, char b, int c) : a(a), b(b),
    padding_1(0), padding_2(0), padding_3(0),
    c(c) {}
};
//确保c是最后一个填充字节之后的下一个字节。
static_assert(offsetof(test, c) == offsetof(test, padding_3) + 1,
"Object contains intermediate padding");
//确保没有尾填充。
static_assert(sizeof(test) == offsetof(test, c) + sizeof(int),
"Object contains trailing padding");
// 安全复制字节到用户空间
extern int copy_to_user(void *dest, void *src, std::size_t size);
void do_stuff(void *usr_buf) {
    test arg{1, 2, 3};
    copy_to_user(usr_buf, &arg, sizeof(arg));
}
```

参考文献

DSL55-CPP

DCL56-CPP. 避免循环初始化静态对象

作者：罗胜金10041900 评审人： 刘光聪10209986

本规则要求，
第一，在静态变量声明的初始化期间不要重新进入函数；
第二，在无法保证初始化顺序时，禁止静态对象之间在初始化时存在相互依赖关系。

不合规代码

以下代码，试图使用缓存实现高效的阶乘函数。由于静态局部数组 cache 的初始化涉及递归，即使不是无限递归，该函数也可能导致不可预知结果。

```
#include <stdexcept>
int fact(int i) noexcept(false) {
    if (i < 0) {
    // 负因子是未定义的
    throw std::domain_error("i must be >= 0");
    }

    static const int cache[] = {
    fact(0), fact(1), fact(2), fact(3), fact(4), fact(5),
    fact(6), fact(7), fact(8), fact(9), fact(10), fact(11),
    fact(12), fact(13), fact(14), fact(15), fact(16)
    };

    if (i < (sizeof(cache) / sizeof(int))) {
    return cache[i];
    }

    return i > 0 ? i * fact(i - 1) : 1;
}
```

在Microsoft Visual Studio 2015和GCC 6.1.0中，在 cache 的递归初始化中，即使以线程安全的方式初始化静态变量，也出现死锁。

合规代码

以下代码，没有初始化本地静态数组 cache ，而是依赖于零初始化来确定数组中每个成员是否已经被分配值，如果没有，则递归计算其值。

```
#include <stdexcept>
int fact(int i) noexcept(false) {
    if (i < 0) {
    // 负因子是未定义的
    throw std::domain_error("i must be >= 0");
    }

    // 使用延迟初始化的cache
    static int cache[17];
    if (i < (sizeof(cache) / sizeof(int))) {
        if (0 == cache[i]) {
        cache[i] = i > 0 ? i * fact(i - 1) : 1;
        }
        return cache[i];
    }

    return i > 0 ? i * fact(i - 1) : 1;
}
```

不合规代码

以下代码， `file1.cpp` 中 `numWheels` 的值依赖于 `c` 的初始化。然而，由于 `c` 定义在另一个编译单元中(`file2.cpp`)，不能保证 `c` 初始化先于 `numWheels` ，这会导致不可预知结果。

```
// file.h
#ifndef FILE_H
#define FILE_H
class Car {
    int numWheels;
public:
    Car() : numWheels(4) {}
    explicit Car(int numWheels) : numWheels(numWheels) {}
    int get_num_wheels() const { return numWheels; }
};
#endif // FILE_H
// file1.cpp
#include "file.h"
#include <iostream>
extern Car c;
int numWheels = c.get_num_wheels();
int main() {
```

```
    std::cout << numWheels << std::endl;
}
// file2.cpp
#include "file.h"
Car get_default_car() { return Car(6); }
Car c = get_default_car();
```

以上代码的运行结果，依赖于编译链接顺序。例如，在X86 Linux上使用Clang 3.8.0，输入命令 `clang++ file1.cpp file2.cpp && ./a.out` 会输出 `0` ，输入命令 `clang++ file2.cpp file1.cpp &&./a.out` 会输出 `6` 。

合规代码

以下代码，静态变量 `numWheels` 移到函数体中，保证 `numWheels` 的初始化顺序在 `c` 之后。

```
// file.h
#ifndef FILE_H
#define FILE_H
class Car {
    int numWheels;
public:
    Car() : numWheels(4) {}
    explicit Car(int numWheels) : numWheels(numWheels) {}
    int get_num_wheels() const { return numWheels; }
};
#endif // FILE_H
// file1.cpp
#include "file.h"
#include <iostream>
int &get_num_wheels() {
    extern Car c;
    static int numWheels = c.get_num_wheels();
    return numWheels;
}
int main() {
    std::cout << get_num_wheels() << std::endl;
}
// file2.cpp
#include "file.h"
Car get_default_car() { return Car(6); }
Car c = get_default_car();
```

参考文献

DSL56-CPP

DCL57-CPP.不要让异常逃出析构函数或内存释放函数

作者：罗胜金10041900 评审人： 刘光聪10209986

禁止在析构函数、 `delete` 操作符或 `delete[]` 操作符中抛出异常，否则可能导致不可预知结果。

不合规代码

以下代码，析构函数不满足隐式 `noexcept` 保证，它可能抛出异常，引起不可预知结果。

```
#include <stdexcept>
class S {
    bool has_error() const;
public:
    ~S() noexcept(false) {
        // 正常处理
        if (has_error()) {
            throw std::logic_error("Something bad");
        }
    }
};
```

不合规代码(function-try-block)

以下代码，类 `Bad` 的析构函数可抛出异常。该类违反了本规则，假定它不能修改成符合本规则。

```
//假设这个类是由第三方提供的，它不是可以被用户修改的东西。
class Bad {
    ~Bad() noexcept(false);
};
```

为了安全使用 `Bad` 类， `SomeClass` 的析构函数尝试捕捉并处理从 `Bad` 析构函数抛出的异常。

```
class SomeClass {
    Bad bad_member;
public:
    ~SomeClass(){
        try {
        // ...
        } catch(...) {
        // 处理从Bad抛出的异常
        }
    }
};
```

然而，根据C++标准，这里捕获的异常，将从 `SomeClass` 析构函数中逃脱，因为它在到达 `function-try-block` 的末尾时将被隐式重新抛出。

合规代码

析构函数应该仅调用不抛出异常的操作，或者处理所有异常，不能重新抛出异常（即使是隐式地）。下面代码的析构函数有一个显式返回，防止流程到达异常处理程序的末尾。

```
class SomeClass {
    Bad bad_member;
public:
    ~SomeClass()
    try {
    // ...
    } catch(...) {
        // 捕获从成员对象或基类子对象的不合规析构函数抛出的异常。
        //注意：析构函数结尾fuction-try-block会导致已捕获的异常被重新抛出，但是一个显式的返回语句将阻止这种情况的发生。
        return;
    }
};
```

不合规代码

下面代码中，全局内存释放函数被声明为 `noexcept(false)` ，允许抛出异常，可能导致不可预知结果。

```
#include <stdexcept>
bool perform_dealloc(void *);
```

```
void operator delete(void *ptr) noexcept(false) {
    if (perform_dealloc(ptr)) {
        throw std::logic_error("Something bad");
    }
}
```

合规代码

下面代码，不会在释放内存失败时抛出异常，而是会尽可能优雅地失败。

```
#include <cstdlib>
#include <stdexcept>
bool perform_dealloc(void *);
void log_failure(const char *);
void operator delete(void *ptr) noexcept(true) {
    if (perform_dealloc(ptr)) {
        log_failure("Deallocation of pointer failed");
        std::exit(1); //失败，但仍会调用析构函数
    }
}
```

参考文献

DCL57-CPP

DCL58-CPP. 不要修改标准命名空间

作者：罗胜金10041900 评审人： 刘光聪10209986

不要向标准命名空间std、posix及其包含的命名空间中添加声明或定义。

不合规代码

以下代码，将 x 的声明添加到命名空间std中，会导致不可预知结果。

```
namespace std {
    int x;
}
```

合规代码

以下代码，将 x 的声明放到一个非标准命名空间nonstd中，防止 x 与其他全局标识符冲突。

```
namespace nonstd {
    int x;
}
```

参考文献

DCL58-CPP

DCL59-CPP. 不要在头文件中定义匿名的命名空间

作者：罗胜金10041900 评审人： 刘光聪10209986

如果在某个头文件中定义匿名的命名空间，假设此匿名命名空间中定义了变量或函数，则所有包含这个头文件的编译单元中，将各自生成这些变量或函数的唯一实例，引起意外错误。

不合规代码

以下代码，变量 v 定义在头文件的匿名空间中，导致a.cpp和b.cpp各自生成一个不同的 v 实例，引起意外结果。

```
// a.h
#ifndef A_HEADER_FILE
#define A_HEADER_FILE
namespace {
    int v;
}
#endif // A_HEADER_FILE
// a.cpp
#include "a.h"
#include <iostream>
void f() {
    std::cout << "f(): " << v << std::endl;
    v = 42;
    // ...
```

```
}
// b.cpp
#include "a.h"
#include <iostream>
void g() {
    std::cout << "g(): " << v << std::endl;
    v = 100;
}
int main() {
    extern void f();
    f(); // Prints v, sets it to 42
    g(); // Prints v, sets it to 100
    f();
    g();
}
```

当程序执行时，打印如下：

```
f(): 0
g(): 0
f(): 42
g(): 100
```

合规代码

以下代码， v 仅在一个编译单元中定义，但对于所有编译单元可见，其行为符合预期。

```
// a.h
#ifndef A_HEADER_FILE
#define A_HEADER_FILE
extern int v;
#endif // A_HEADER_FILE
// a.cpp
#include "a.h"
#include <iostream>
int v; // Definition of global variable v
void f() {
    std::cout << "f(): " << v << std::endl;
    v = 42;
    // ...
}
// b.cpp
```

```
#include "a.h"
#include <iostream>
void g() {
    std::cout << "g(): " << v << std::endl;
    v = 100;
}
int main() {
    extern void f();
    f(); // Prints v, sets it to 42
    g(); // Prints v, sets it to 100
    f(); // Prints v, sets it back to 42
    g(); // Prints v, sets it back to 100
}
```

程序执行结果如下：

```
f(): 0
g(): 42
f(): 100
g(): 42
```

不合规代码

如下代码，在头文件的匿名空间中定义一个f()函数，导致最终生成多个不同的 `f()` 定义，链接时间增加，执行文件变大，性能降低。

```
// a.h
#ifndef A_HEADER_FILE
#define A_HEADER_FILE
namespace {
    void f() { /* ... */ }
}
#endif // A_HEADER_FILE
// a.cpp
#include "a.h"
// ...
// b.cpp
#include "a.h"
// ...
```

合规代码

如下代码， `f()` 没有使用匿名的命名空间，而是定义为内联函数。内联函数在包含它们的所有编译单元中定义相同，编译器仅生成函数的单个实例。

```
// a.h
#ifndef A_HEADER_FILE
#define A_HEADER_FILE
inline void f() { /* ... */ }
#endif // A_HEADER_FILE
// a.cpp
#include "a.h"
// ...
// b.cpp
#include "a.h"
// ...
```

参考文献

DCL59-CPP

Rule 02. 表达式

EXP50-CPP. 不要依赖求值顺序

作者： 李永顺10110636 评审人：罗胜金10041900

C++表达式或对象初始化，不依赖于求值顺序，以免带来不确定性，这个规则意味着，下面的声明具有确定的行为。

```
i = i + 1;
a[i] = i;
```

而如下的声明则没有

```
// i is modified twice in the same full expression
i = ++i + 1;
// i is read ot
```

常见的一些可能相互依赖影响的场景：

- 在逻辑 `&&` 或者 `||` 表达式中，如果对第二个表达式求值，则第一个表达式可能会对第二个表达式有影响
- 在条件表达式 `?` `:` 中，第一个表达式可能对第二个、第三个表达式的影响
- 当对初始化列表求值时，与每个初始化子句相关联的值计算可能会对排在其后面初始化子句有影响
- 不确定的函数执行顺序，函数执行的顺序不同，运行结果不同，尤其是在全局初始化阶段尤为明显

不合规代码

在这个不合规的代码示例中， `i` 以不确定的顺序被求值不止一次，因此表达式的行为是未定义的。

```
void f(int i, const int *b) {
    int a = i + b[++i];
    // ...
}
```

合规代码

这些示例独立于操作数的求值顺序，并且每个只有一种解释方式。

```
void f(int i, const int *b) {
    ++i;
    int a = i + b[i];
    // ...
}
```

```
void f(int i, const int *b) {
    int a = i + b[i + 1];
    ++i;
    // ...
}
```

不合规代码

此不合规代码示例，展示了函数参数的求值顺序未确定所带来的不确定行为。

```
extern void c(int i, int j);
int glob;
```

```
int a() {
    return glob + 10;
}
int b() {
    glob = 42;
    return glob;
}
void f() {
    c(a(), b());
}
```

未指定 a() 和 b() 的调用顺序，唯一可以确认的是在调用 c() 之前调用 a() 和 b() 。 如果 a() 或 b() 在计算它们的返回值时依赖于执行顺序，就像在这个例子中一样，传递给 c() 的参数可能在不同的编译器计算结果不同。

合规代码

合规代码中， a() 和 b() 的求值顺序是固定的，所以不会产生不确定行为。

```
extern void c(int i, int j);
int glob;
int a() {
    return glob + 10;
}
int b() {
    glob = 42;
    return glob;
}
void f() {
    int a_val, b_val;
    a_val = a();
    b_val = b();
    c(a_val, b_val);
}
```

参考文献

- EXP50-CPP. Do not depend on the order of evaluation for side effects

EXP51-CPP. 不要通过不正确类型指针删除一个数组

作者： 李永顺10110636 评审人：罗胜金10041900

如果通过基类指针删除含有虚函数的子类数组，会导致无法预测的结果。

不合规代码

本示例中创建了一个 `Derived` 类型的数组对象，保存为 `Base *` 。尽管 `Base::~Base()` 被声明为虚的，仍然会导致无法预测的结果。

```
struct Base {
    virtual ~Base() = default;
};
struct Derived final : Base {};
void f() {
    Base *b = new Derived[10];
    // ...
    delete [] b;
}
```

合规代码

在下面这个合规解决方案中， `b` 的静态类型是 `Derived *` ，这样当引用到这个数组对象和删除这个指针的时候就不会产生未定义的行为。

```
struct Base {
    virtual ~Base() = default;
};
struct Derived final : Base {};
void f() {
    Derived *b = new Derived[10];
    // ...
    delete [] b;
}
```

参考文献

- EXP51-CPP. Do not delete an array through a pointer of the incorrect type

EXP52-CPP. 不要依赖不求值的操作数

作者： 李永顺10110636 评审人：罗胜金10041900

C++ 中，如下表达式不对操作数求值，包括：

> *sizeof()* ， *typeid()* ， *noexcept()* ， *decltype()* ， *declval()*

不要在该类表达式中改变操作数的值。

不合规代码（sizeof）

如下代码， a++ 这个表达式不会生效

```
#include <iostream>
    void f() {
    int a = 14;
    int b = sizeof(a++);
    std::cout << a << ", " << b << std::endl;
}
```

因此， a 的值在 b 初始化之后还是14.

合规代码（sizeof）

如下代码，变量 a 的值在 sizeof() 操作符外面 ++

```
#include <iostream>
    void f() {
    int a = 14;
    int b = sizeof(a);
    ++a;
    std::cout << a << ", " << b << std::endl;
}
```

不合规代码（decltype）

如下代码， i++ 这个表达式在 decltype() 说明符中不被求值。

```
#include <iostream>
void f() {
    int i = 0;
    decltype(i++) h = 12;
```

```
    std::cout << i;
}
```

因此， `i` 的值仍然是0。

合规代码（decltype）

如下代码， `i` 在 `decltype` 说明符外面 `++` ，与预期结果相同

```
#include <iostream>
void f() {
    int i = 0;
    decltype(i) h = 12;
    ++i;
    std::cout << i;
}
```

参考文献

- EXP52-CPP. Do not rely on side effects in unevaluated operands

EXP53-CPP. 不要读取未初始化的内存

作者： 曾亮亮10091330 评审人：罗胜金10041900

在对象初始化之前读取对象的值，得到的是随机值。

常见场景：

- 函数内局部变量未初始化
- 类的成员变量未初始化

不合规代码

在这个不合规代码中，未初始化的局部变量将作为表达式的一部分进行计算，打印出来的值为随机值。

```
#include <iostream>
void f() {
    int i;
```

```
    std::cout << i;
}
```

合规代码

在这个合规解决方案中，对象在打印其值之前进行初始化。

```
#include <iostream>
void f() {
    int i = 0;
    std::cout << i;
}
```

不合规代码

在这个不合规代码示例中， `int *` 对象由一个 `new-expression` 分配，但它指向的内存不会被初始化。打印指针值是明确定义的，但它指向的值是随机的。

```
#include <iostream>
void f() {
    int *i = new int;
    std::cout << i << ", " << *i;
}
```

合规代码

在这个合规解决方案中，在打印其值之前，将其值初始化为12。

```
#include <iostream>
void f() {
    int *i = new int(12);
    std::cout << i << ", " << *i;
}
```

可以在分配的类型之后加上括号或花括号，来完成由 `new-expression` 生成的对象初始化，如果初始化不提供默认值，则会用零来初始化对象，如下面的代码所示：

```
int *i = new int(); // zero-initializes *i
int *j = new int{}; // zero-initializes *j
int *k = new int(12); // initializes *k to 12
```

```
int *l = new int{12}; // initializes *l to 12
```

不合规代码

在这个不合规代码示例中，类成员变量 `c` 没有被默认构造函数中的 `ctor-initializer` 显式初始化，虽然局部变量 `s` 已经被默认初始化，但在调用 `S::f()` 中使用的 `c` 是未初始化的。

```
class S {
    int c;
    public:
    int f(int i) const { return i + c; }
};
void f() {
    S s;
    int i = s.f(10);
}
```

合规代码

在合规代码中，类 `S` 给出了一个默认构造函数初始化了类成员变量 `c` 。

```
class S {
    int c;
    public:
    S() : c(0) {}
    int f(int i) const { return i + c; }
};
void f() {
    S s;
    int i = s.f(10);
}
```

参考文献

- Do not read uninitialized memory

EXP54-CPP. 不要访问在其生命周期之外的对象

作者： 曾亮亮10091330 评审人：罗胜金10041900

每个对象都有其生命周期，在对象初始化之后、且在对象析构之前可以正常使用。

不合规代码

在这个不合规代码示例中，对象的指针在对象的生命周期开始之前，被用来调用指向其对象的非静态成员函数，会导致未定义的行为。

```
struct S {
    void mem_fn();
};
void f() {
    S *s;
    s->mem_fn();
}
```

合规代码

在这个合规解决方案中，在调用 `S::mem_fn()` 函数之前，确保对象的指针内存已分配。

```
struct S {
    void mem_fn();
};
void f() {
    S *s = new S;
    s->mem_fn();
    delete s;
}
```

不合规代码

在这个不合规代码示例中，对象指针在其生命周期结束后被隐式转换为一个虚拟基类，会导致未定义的行为。

```
struct B {};
struct D1 : virtual B {};
struct D2 : virtual B {};
struct S : D1, D2 {};
void f(const B *b) {}
void g() {
```

```
    S *s = new S;
    // Use s
    delete s;
    f(s);
}
```

NOTES： 尽管实际上 `f()` 并没有使用这个对象，但它作为参数传递给 `f()` 足以触发未定义的行为。

不合规代码

在这个不合规代码示例中， `f()` 返回局部变量的地址。当将生成的指针传递给 `h()` 时，对 `i` 的lvalue-to-rvalue的转换会导致未定义的行为。

```
int *g() {
    int i = 12;
    return &i;
}
void h(int *i);
void f() {
    int *i = g();
    h(i);
}
```

NOTES： 从函数返回局部对象的指针，一些编译器会生成诊断信息。

合规代码

在这个合规解决方案中，从 `g()` 返回的局部变量具有静态存储属性，在 `f()` 内其生命周期都是有效的。

```
int *g() {
    static int i = 12;
    return &i;
}
void h(int *i);
void f() {
    int *i = g();
    h(i);
}
```

不合规代码

`std::initializer_list<>` 对象是从初始化列表构造的，就类似于实现分配了一个临时数组并将其传递给 `std::initializer_list<>` 构造函数，这个临时数组与其他临时对象具有相同的生命周期。
这个不合规代码示例中， `std::initializer_list<int>` 类型的成员变量在构造函数的ctor-initializer中被初始化。在这些情况下，这个所谓的临时数组的生命周期在构造函数退出时结束，因此访问 `std::initializer_list<int>` 成员变量的任何元素都会导致未定义的行为。

```
#include <initializer_list>
#include <iostream>
class C {
    std::initializer_list<int> l;
    public:
    C() : l{1, 2, 3} {}
    int first() const { return *l.begin(); }
};
void f() {
    C c;
    std::cout << c.first();
}
```

合规代码

在这个合规解决方案中， `std::initializer_list<int>` 的成员变量被 `std::vector<int>` 代替，它将初始化列表中的元素复制到容器，而不是依赖于对临时数组的悬挂引用。

```
#include <iostream>
#include <vector>
class C {
    std::vector<int> l;
    public:
    C() : l{1, 2, 3} {}
    int first() const { return *l.begin(); }
};
void f() {
    C c;
    std::cout << c.first();
}
```

不合规代码

在这个不合规代码示例中， `lambda` 对象存储在函数对象中，在后面调用（执行 `lambda` ）以获取对值的常量引用。 `lambda` 对象返回一个 `int` 值，然后存储在一个临时的 `int` 对象中，该对象被绑定到函数对象 `const int&` 的返回值类型。但是，临时对象的生命周期，不会超过函数对象被调用的返回值，当访问结果值时会导致未定义的行为。

```
#include <functional>

void f() {
  auto l = [](const int &j) { return j; };
  std::function<const int&(const int &)> fn(l);

  int i = 42;
  int j = fn(i);
}
```

合规代码

这个合规解决方案中， `std::function` 对象返回一个 `int` 而不是 `const int&` ，确保该值被赋值，而不是绑定到一个临时引用。另一种解决方案是直接调用 `lambda` ，而不是通过 `std::function<>` 对象。

```
#include <functional>

void f() {
  auto l = [](const int &j) { return j; };
  std::function<int(const int &)> fn(l);

  int i = 42;
  int j = fn(i);
}
}
```

不合规代码

在这个不合规代码示例中，不会调用自动变量 `s` 的构造函数，因为 `goto` 声明的存在，

导致不会走到局部变量的声明。由于构造函数没有被调用， `s` 的生命周期也就没有开始。所以，调用未构造的对象s的方法 `s::f()` 会导致未定义的行为。

```
class S {
  int v;

public:
  S() : v(12) {} // Non-trivial constructor

  void f();
};

void f() {

  // ...

  goto bad_idea;

  // ...

  S s; // Control passes over the declaration, so initialization does not take place
.

  bad_idea:
    s.f();
}
```

合规解决方案

这个合规解决方案确保在执行本次跳转之前已正确初始化。

```
class S {
  int v;

public:
  S() : v(12) {} // Non-trivial constructor

  void f();
};

void f() {
  S s;
```

```
  // ...

  goto bad_idea;

  // ...

  bad_idea:
    s.f();
}
```

参考文献

- EXP54-CPP. Do not access an object outside of its lifetime

EXP55-CPP. 不要通过cv-unqualified类型访问一个cv-qualified对象

作者： 曾亮亮10091330 评审人：罗胜金10041900

不要无视一个 `const` 限定来尝试修改生成的对象， `const` 限定符意味着API设计者不打算修改对象，尽管它可能是可修改的。
不要无视一个 `volatile` 限定， `volatile` 限定符意味着API设计者意图以编译器未知的方式访问对象。

不合规代码

在这个不合规代码示例中，函数 `g()` 传递一个 `const int&` 类型的参数，然后这个参数被转换为 `int&` 类型并进行修改。

```
void g(const int &ci) {
    int &ir = const_cast<int &>(ci);
    ir = 42;
}
void f() {
    const int i = 4;
    g(i);
}
```

合规代码

这个合规解决方案中， g() 函数传入一个 int & 类型的参数，而调用者需要传递一个 int 类型的参数从而可以修改它。

```
void g(int &i) {
    i = 42;
}
void f() {
    int i = 4;
    g(i);
}
```

不合规代码

在这个不合规代码示例中， s 具有 volatile 限定符，用它读取 g() 中的值，会导致未定义的行为。

```
#include <iostream>
struct S {
    int i;
    S(int i) : i(i) {}
};
void g(S &s) {
    std::cout << s.i << std::endl;
}
void f() {
    volatile S s(12);
    g(const_cast<S &>(s));
}
```

合规代码

在这个合规解决方案中，认为 s 必须为 volatile 类型，所以 g() 修改为接受一个 volatile S & 类型的参数。

```
#include <iostream>
struct S {
    int i;
```

```
    S(int i) : i(i) {}
};
void g(volatile S &s) {
    std::cout << s.i << std::endl;
}
void f() {
    volatile S s(12);
    g(s);
}
```

参考文献

- EXP55-CPP. Do not access a cv-qualified object through a cv-unqualified type

EXP57-CPP. 不要强制转换或删除指向不完整类的指针

作者： 曾亮亮10091330 评审人：罗胜金10041900

引用不完全类类型的对象，也称为前向声明，是一种常见的做法。
但是以下两种情况，不能使用不完全类类型：

- 尝试删除指向不完整类类型的对象的指针
- 不要尝试转换指向不完整类类型的对象的指针

不合规代码

在这个不合规代码示例中，这个类试图实现 `pimpl` 的常规语法，但是删除了一个指向不完整的类类型的指针。如果 `Body` 具有一个nontrivial析构函数，则会导致未定义的行为。

```
class Handle {
    class Body *impl; // Declaration of a pointer to an incomplete class
    public:
    ~Handle() { delete impl; } // Deletion of pointer to an incomplete class
    // ...
};
```

合规代码（ `delete` ）

在这个合规解放方案中，将 `impl` 的删除移动到定义 `Body` 代码中，使其删除时知道该指

针的完整类型。

```
class Handle {
    class Body *impl; // Declaration of a pointer to an incomplete class
    public:
    ~Handle();
    // ...
};
// Elsewhere
class Body { /* ... */ };
Handle::~Handle() {
    delete impl;
}
```

不合规代码

指针向下转换（将指向基类的指针转换为指向派生类的指针）可能需要以固定值来调整指针的地址，只有当类继承的布局已知时才能确定该值。
这个不合规代码中， `f()` 从 `get_d()` 中得到一个完整类型B的多态指针，该指针后面被转换为不完整的类型D的指针，然后传递给 `g()` 。强制转换为指向派生类的指针，可能无法正确调整生成的指针，当通过调用 `d->do_something()` 解引用指针时，会导致未定义的行为。

```
// File1.h
class B {
    protected:
    double d;
    public:
    B() : d(1.0) {}
};
// File2.h
void g(class D *);
class B *get_d(); // Returns a pointer to a D object
// File1.cpp
#include "File1.h"
#include "File2.h"
void f() {
    B *v = get_d();
    g(reinterpret_cast<class D *>(v));
}
// File2.cpp
```

```
#include "File2.h"
#include "File1.h"
#include <iostream>
class Hah {
    protected:
    short s;
    public:
    Hah() : s(12) {}
};
class D : public Hah, public B {
    float f;
    public:
    D() : Hah(), B(), f(1.2f) {}
    void do_something() { std::cout << "f: " << f << ", d: " << d << ", s: " << s <<
    std::endl; }
};
void g(D *d) {
    d->do_something();
}
B *get_d() {
    return new D;
}
```

实现细节

当用ClangBB编译时，Definitions#clang3.8，调用函数 `f()` ，不合规代码会打印以下内容。

```
f: 1.89367e-40, d: 5.27183e-315, s: 0
```

类似地，当示例在Microsoft Visual Studio 2015和GCC 6.1.0中运行时，将打印未知值。

合规代码

这个解决方案假设本代码目的是通过使用不完整的类类型来隐藏实现细节。 `g()` 不要求传递一个 `D*` 类型，而是期望传递一个 `B*` 类型。

```
// File1.h -- contents identical.
// File2.h
void g(class B *); // Accepts a B object, expects a D object
class B *get_d(); // Returns a pointer to a D object
```

```
// File1.cpp
#include "File1.h"
#include "File2.h"
void f() {
    B *v = get_d();
    g(v);
}
// File2.cpp
// ... all contents are identical until ...
void g(B *d) {
    D *t = dynamic_cast<D *>(d);
    if (t) {
        t->do_something();
        } else {
            // Handle error
    }
}
B *get_d() {
    return new D;
}
```

参考文献

- EXP57-CPP. Do not cast or delete pointers to incomplete classes

EXP58-CPP. 给va_start传递正确类型的对象

作者： 曾亮亮10091330 评审人：罗胜金10041900

调用va_start()宏时必须小心，其第二个参数值类型有特殊要求：

- 不能使用默认可提升的参数类型
- 不能传引用
- 不能传递有nontrivial复制构造函数的类

不合规代码

在这个不合规代码示例中，传递给 `va_start()` 的对象将经过默认参数提升，从而导致未定义的行为。

```
#include <cstdarg>
```

```
extern "C" void f(float a, ...) {
    va_list list;
    va_start(list, a);
    // ...
    va_end(list);
}
```

合规代码

在这个合规解决方案中，将 f() 的参数由 float 类型改成 double 类型。

```
#include <cstdarg>
extern "C" void f(double a, ...) {
    va_list list;
    va_start(list, a);
    // ...
    va_end(list);
}
```

不合规代码

在这个不合规代码示例中，给va_start()第二个参数传递了引用类型。

```
#include <cstdarg>
#include <iostream>
extern "C" void f(int &a, ...) {
    va_list list;
    va_start(list, a);
    if (a) {
        std::cout << a << ", " << va_arg(list, int);
        a = 100; // Assign something to a for the caller
    }
    va_end(list);
}
```

合规代码

这个合规解决方案中，给 f() 传递一个指针来代替引用。

```
#include <cstdarg>
```

```
#include <iostream>
extern "C" void f(int *a, ...) {
    va_list list;
    va_start(list, a);
    if (a && *a) {
        std::cout << a << ", " << va_arg(list, int);
        *a = 100; // Assign something to *a for the caller
    }
    va_end(list);
}
```

不合规代码

在这个不合规代码示例中，给 `va_start()` 第二个参数传递一个有nontrivial复制构造函数的类（ `std::string` ）。

```
#include <cstdarg>
#include <iostream>
#include <string>
extern "C" void f(std::string s, ...) {
    va_list list;
    va_start(list, s);
    std::cout << s << ", " << va_arg(list, int);
    va_end(list);
}
```

合规解决方案

这个合规解决方案中用 `char *` 代替 `std::string` 来传递，这在所有编译器中都是确定的行为。

```
#include <cstdarg>
#include <iostream>
extern "C" void f(const char *s, ...) {
    va_list list;
    va_start(list, s);
    std::cout << (s ? s : "") << ", " << va_arg(list, int);
    va_end(list);
}
```

参考文献

- EXP58-CPP. Pass an object of the correct type to va_start

EXP61-CPP. lambda对象不能超出其捕获引用对象的生命周期

作者： 李永顺10110636 评审人：罗胜金10041900

*C++11*开始支持lambda表达式，支持在捕捉列表中通过引用（[&]）传递捕捉其副作用域（闭包）中的变量。

如果使用引用方式传递父作用域变量，当lambda对象的作用域超出其捕捉对象的作用域时，执行lambda对象函数运算符，可能导致未定义行为。因此，lambda对象不能超过其捕捉引用类型对象的生命周期。

不合规代码

如下代码中，函数 `g()` 返回lambda对象，当函数 `g()` 调用结束时，变量 `i` 的生命周期已经结束，再调用 `g()()` 时，将由于使用悬挂引用导致未定义行为。

```
auto g() {
    int i = 12;
    return [&] {
        i = 100;
        return i;
    };
}
void f() {
    int j = g()();
}
```

合规代码

如下代码中通过传值([=])方式，避免对变量 `i` 生命周期的依赖。

```
auto g() {
    int i = 12;
    return [=] () mutable {
```

```
        i = 100;
        return i;
    };
}
void f() {
    int j = g()();
}
```

不合规代码

如下代码中， `g(12)` 调用结束，即 `outer()` 调用结束，此时变量 `i` 的生命周期已经结束，在执行 `f()` 时，内部lambda使用生命周期已经结束的外部lambda对象定义的变量，将导致未定义行为。

```
auto g(int val) {
    auto outer = [val] {
        int i = val;
        auto inner = [&] {
            i += 30;
            return i;
        };
        return inner;
    };
    return outer();
}
void f() {
    auto fn = g(12);
    int j = fn();
}
```

合规代码

如下代码中，通过将变量 `i` 复制，避免对外部lambda对象变量 `i` 生命周期的依赖。

```
auto g(int val) {
  auto outer = [val] {
    int i = val;
    auto inner = [i] {
      return i + 30;
    };
    return inner;
```

```
  };
  return outer();
}

void f() {
  auto fn = g(12);
  int j = fn();
}
```

参考文献

- EXP61-CPP. A lambda object must not outlive any of its reference captured objects

EXP62-CPP. 不要访问对象中非对象值部分的bits

作者： 李永顺10110636 评审人：罗胜金10041900

C++中窄字符类型（ `char, signed char, unsigned char`)及特定平台的整数类型，访问对象值表示的任何bits都是明确定义的，即可以通过它的bits来访问或者修改对象。

其他类型（比如类），其对象可能由构成对象值的bits及其他bits构成（例如：虚表指针，填充字节，不具有访问权限的字段等），对该类对象，访问非对象值外的bits，在不同编译器或平台下，可能导致不确定行为。

不合规代码

如下代码中，通过 `std::memcmp` 比较两个S对象，C++ 标准规定，类可以通过填充数据保证在存储器中正确对齐，填充内容及填充大小与编译器相关。当比较内容为非S对象值bits时，可能导致不确定行为。

```
#include <cstring>
struct S {
    unsigned char buffType;
    int size;
};
void f(const S &s1, const S &s2) {
    if (!std::memcmp(&s1, &s2, sizeof(S))) {
        // ...
    }
}
```

合规代码

在如下代码中，通过重载操作符 `==()` 来进行对象值的比较。

```
struct S {
    unsigned char buffType;
    int size;
    friend bool operator==(const S &lhs, const S &rhs) {
        return lhs.buffType == rhs.buffType &&
        lhs.size == rhs.size;
    }
};
void f(const S &s1, const S &s2) {
    if (s1 == s2) {
        // ...
    }
}
```

不合规代码

如下代码中， `std :: memset()` 用于初始化对象s。 由于类中存在虚函数，编译器会在对象s中增加虚表指针， `std::memset` 会把虚表指针也同时初始化，调用对象的虚方法时，可能导致不确定行为。

```
#include <cstring>
struct S {
    int i, j, k;
    // ...
    virtual void f();
};
void f() {
    S *s = new S;
    // ...
    std::memset(s, 0, sizeof(S));
    // ...
    s->f(); // undefined behavior
}
```

合规代码

如下代码中，S的数据成员被通过 `clear()` 显式初始化，而不是调用 `std::memset()` 初

始化。

```
struct S {
    int i, j, k;
    // ...
    virtual void f();
    void clear() { i = j = k = 0; }
};
void f() {
    S *s = new S;
    // ...
    s->clear();
    // ...
    s->f(); // ok
}
```

参考文献

- EXP62-CPP. Do not access the bits of an object representation that are not part of the object's value representation

EXP63-CPP. 不要依赖移动对象的值

作者： 李永顺10110636 评审人：罗胜金10041900

C++11 支持移动语义，可以避免大块内存拷贝造成的性能开销，但对移动语义对象的依赖，可能由于对象处于不确定状态，而造成不确定的行为。

如下STL函数保证在离开移动对象时处于确定状态：

> *std::unique_ptr, std::shared_ptr, std::weak_ptr, std::basic_ios, std::basic_filebuf, std::thread, std::unique_lock, std::shared_lock, std::promise, std::future, std::shared_future, std::packaged_task*

但是部分STL算法，从容器中删除元素后，不会更改容器的大小，而是会返回一个分区 `Forward Iterator` ，该分区之前元素都为有效元素，其后元素为无效元素（内存存在，值不确定）。访问该迭代器指向的元素会导致不确定行为。通常情况下，使用该类算法需要使用 `erase-remove` 来调整其大小。

不合规代码

如下代码实例中，期望通过 std::string 类型的右值引用，打印整数0~9到标准输出流。由于右值引用不会清除 s 对象内存，导致打印出非期望的输出。

```
#include <iostream>
#include <string>
void g(std::string &&v) {
    std::cout << v << std::endl;
}
void f() {
    std::string s;
    for (unsigned i = 0; i < 10; ++i) {
        s.append(1, static_cast<char>('0' + i));
        g(std::move(s));
    }
}
```

基于libc ++的Clang 3.7编译器，打印如下：

```
0
0 1
0 1 2
0 1 2 3
0 1 2 3 4
0 1 2 3 4 5
0 1 2 3 4 5 6
0 1 2 3 4 5 6 7
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8 9
```

合规代码

如下代码中， std :: string 对象为循环内部临时对象，被初始化为确定值。

```
#include <iostream>
#include <string>
void g(std::string &&v) {
    std::cout << v << std::endl;
}
void f() {
    for (unsigned i = 0; i < 10; ++i) {
        std::string s(1, static_cast<char>('0' + i));
```

```
        g(std::move(s));
    }
}
```

不合规代码

如下代码中，期望从给定容器中删除值为42的元素。 然后将容器的内容打印到标准输出流。 但是使用 std::remove 不会改变容器大小，接着遍历容器，将导致未确定行为。

```
#include <algorithm>
#include <iostream>
#include <vector>
void f(std::vector<int> &c) {
    std::remove(c.begin(), c.end(), 42);
        for (auto v : c) {
        std::cout << "Container element: " << v << std::endl;
    }
}
```

合规代码

如下代码中，通过对迭代器 i 分区限制（迭代器分区 e 之前元素为有效元素），保证行为的正常。

```
#include <algorithm>
#include <iostream>
#include <vector>
void f(std::vector<int> &c) {
    auto e = std::remove(c.begin(), c.end(), 42);
    for (auto i = c.begin(); i != c.end(); i++) {
        if (i < e) {
            std::cout << *i << std::endl;
        }
    }
}
```

合规代码

如下代码中，先通过 std::erase 把 std::remove 元素擦除，保证后续遍历迭代器元素的有效性。

```
#include <algorithm>
#include <iostream>
#include <vector>
void f(std::vector<int> &c) {
    c.erase(std::remove(c.begin(), c.end(), 42), c.end());
    for (auto v : c) {
        std::cout << "Container element: " << v << std::endl;
    }
}
```

参考文献

- EXP63-CPP. Do not rely on the value of a moved-from object

Rule 03. 整型

INT50-CPP. 不要做超过枚举值范围的枚举类型转换

作者： 李道春10073354 刘涛10054087 评审人：傅锦华10108953

C+ + 中枚举有两种形式：基础类型固定的领域枚举，基础类型可能被修改的非领域枚举。可以通过枚举类型表示值的范围，也可以包括未指定枚举值的范围。
为了避免对未确定的值进行操作，被转换的算术值必须在枚举可以表示的值得范围内。当动态检查超出范围值时，必须在转换表达式之前进行检查。

不合规的代码示例（边界检查）

本例中尝试检查给定值是否在枚举值可接受的范围内。但是它是在转换为枚举类型后执行的，可能无法表示给定的整数值。在二进制补码系统上，由枚举类型表示值的有效范围是[0..3]，因此如果该范围外的值传递给f()，则向枚举类型转换将导致未指定的值，并且在if语句中使用该值会导致未定义行为。

```
enum EnumType {
  First,
  Second,
  Third
};
```

```
void f(int intVar) {
  EnumType enumVar = static_cast<EnumType>(intVar);

  if (enumVar < First || enumVar > Third) {
    // 捕获错误
  }
}
```

合规的代码示例（边界检查）

此解决方案的检查在执行转换之前通过枚举类型表示该值，以确保转换不会导致未指定的值。它通过将转换的值限制为具有特性枚举值的值来实现。

```
enum EnumType {
  First,
  Second,
  Third
};

void f(int intVar) {
  if (intVar < First || intVar > Third) {
    // 捕获错误
  }
  EnumType enumVar = static_cast<EnumType>(intVar);
}
```

合规的代码示例（领域枚举）（C++11支持）

此解决方案使用领域枚举，默认情况下是具有固定的底层使用int类型，允许将参数中的任何值转换为有效的枚举值。它不会将转换的值限制为具有特定枚举器值的转换值，但可以按照之前的合规解决方案所示进行操作。

```
enum class EnumType {
  First,
  Second,
  Third
};

void f(int intVar) {
  EnumType enumVar = static_cast<EnumType>(intVar);
}
```

合规的代码示例（固定的非领域枚举）（C++11支持）

与前一个合规的代码示例类似，此解决方案使用未限定的枚举，但提供了一个固定的底层int类型，允许将参数中的任何值转换为有效的枚举值。

```
enum EnumType : int {
  First,
  Second,
  Third
};

void f(int intVar) {
  EnumType enumVar = static_cast<EnumType>(intVar);
}
```

虽然类似于之前的合规解决方案，但是该解决方案与不合规的代码示例之间的不同之处在于枚举器值以代码表示，允许隐式转换。 先前的合规解决方案需要嵌套的名称说明符来标识枚举器（例如， `EnumType::First` ），并且不会将枚举器值隐式转换为 `int` 。 与不合规的代码示例一样，此解决方案不允许嵌套的名称说明符，并将枚举器值隐式转换为 `int` 。

参考文献

INT50-CPP. Do not cast to an out-of-range enumeration value

Rule 04. 容器

CTR50-CPP. 保证容器的下标和迭代器在有效的范围内 (刘涛)

作者： 李道春10073354 刘涛10054087 评审人：傅锦华10108953

确保数组的下标在数组的边界范围内是程序员的责任。同样的，当使用标准模板库的vector等容器时，程序员需要确保下标在容器的边界范围内。

不合规代码

在本例中，insert_in_table函数有两个int类型参数pos和value，两者的值可以来自不被信任的数据源。该函数通过tableSize校验了pos的值，保证了数组不会超过上限，但它没有校验数组的下限。由于pos定义为int类型，该实参可以是一个负数，结果导致table数组写越界。

```
#include <cstddef>

void insert_in_table(int *table, std::size_t tableSize, int pos, int value) {
  if (pos >= tableSize) {
    // Handle error
    return;
  }
  table[pos] = value;
}
```

合规代码(size_t)

在本例中，参数pos定义为size_t，确保传递一个负数时校验会失败，当把负数转换为无符号整数时。

```
#include <cstddef>
void insert_in_table(int *table, std::size_t tableSize, std::size_t pos, int value)
{
  if (pos >= tableSize) {
    // Handle error
    return;
  }
  table[pos] = value;
}
```

不合规代码(std:vector)

在本例中，一个vector对象通过引用使用。该函数校验了pos的范围以确保pos没有超过上限，但是pos定义为long类型，该实参可以传递一个负数。在系统中std::vector::size_type是最终实现为无符号整数unsigned int(例如VS2013),一般的整数类型转换会把无符号整数转换为有符号整数。如果pos是一个负数，该检验失效，当用作下标时被解释为一个很大的无符号整数，结果导致vector对象写越界。

```
#include <vector>
```

```
void insert_in_table(std::vector<int> &table, long long pos, int value) {
  if (pos >= table.size()) {
    // Handle error
    return;
  }
  table[pos] = value;
}
```

合规代码(std:vector, size_t)

在本例中，参数pos定义为std::size_t, 确保传递一个负数时会把负数转换为无符号整数，这样校验边界就会发挥作用。

```
#include <vector>

void insert_in_table(std::vector<int> &table, std::size_t pos, int value) {
  if (pos >= table.size()) {
    // Handle error
    return;
  }
  table[pos] = value;
}
```

合规代码(std:vector::at())(C++11支持)

在本例中，访问vector是由at方法来实现的。该方法进行了边界校验，如果pos是一个无效的索引值，则会抛出一个std::out_of_range异常。insert_in_table函数需要声明为noexcept(false)。

```
#include <vector>

void insert_in_table(std::vector<int> &table, std::size_t pos, int value) noexcept(f
alse) {
  table.at(pos) = value;
}
```

不合规代码(iterator)

在本例中，f_imp函数的参数e表示容器的末尾，而b则是容器的起始。然而，b有可能不在容器的有效索引范围内。例如，如果容器为空，b和e都是不正确的引用。

```
#include <iterator>

template <typename ForwardIterator>
void f_imp(ForwardIterator b, ForwardIterator e, int val, std::forward_iterator_tag)
 {
  do {
    *b++ = val;
  } while (b != e);
}
template <typename ForwardIterator>
void f(ForwardIterator b, ForwardIterator e, int val) {
  typename std::iterator_traits<ForwardIterator>::iterator_category cat;
  f_imp(b, e, val, cat);
}
```

合规代码

在本例中，在使用iterator之前校验它的有效性。

```
#include <iterator>

template <typename ForwardIterator>
void f_imp(ForwardIterator b, ForwardIterator e, int val, std::forward_iterator_tag)
 {
  while (b != e) {
    *b++ = val;
  }
}
template <typename ForwardIterator>
void f(ForwardIterator b, ForwardIterator e, int val) {
  typename std::iterator_traits<ForwardIterator>::iterator_category cat;
  f_imp(b, e, val, cat);
}
```

参考文献

-CTR50-CPP. Guarantee that container indices and iterators are within the valid range

CTR51-CPP. 用有效的引用、指针或者迭代器来获取容

器中的元素(刘涛)

作者： 李道春10073354 刘涛10054087 评审人：傅锦华10108953

Iterators(迭代器)是一种通用指针，它允许C++程序员采用相同的形式操作不同的数据结构（包括容器）。指针、引用和迭代器共享一个封闭的关系，此关系中，要求引用值可以通过索引、指针和迭代器获取。存在一个风险是当容器发生修改时，原来存储的迭代器、引用和指针可能变得无效。需要确保只使用有效的指针、引用和迭代器来访问容器中的元素。
可能使得迭代器、指针和引用等无效的容器功能，请参见参考文献。

不合规代码

在本例中，在第一次执行insert函数后，pos变得无效，导致遍历会产生无法预料的结果。

```
#include <deque>

void f(const double *items, std::size_t count) {
  std::deque<double> d;
  auto pos = d.begin();
  for (std::size_t i = 0; i < count; ++i, ++pos) {
    d.insert(pos, items[i] + 41.0);
  }
}
```

合规代码

在本例中，变量pos在每次插入之后赋予新值，防止出现未定义的行为。

```
#include <deque>

void f(const double *items, std::size_t count) {
  std::deque<double> d;
  auto pos = d.begin();
  for (std::size_t i = 0; i < count; ++i, ++pos) {
    pos = d.insert(pos, items[i] + 41.0);
  }
}
```

参考文献

- CTR51-CPP. Use valid references, pointers, and iterators to reference elements of a container

CTR52-CPP. 保证库函数不会溢出

作者： 李道春10073354 刘涛10054087 评审人：傅锦华10108953

复制数据到容器时如果数据大小超过容器的容量则会导致缓冲区溢出。要防止该错误，复制到目的容器的数据大小必须被限制在目的容器的大小范围之内。

不合规代码

在本例中，STL容器跟数组类型一样具有相同的缺陷。std::copy()算法提供不进行边界校验的算法实现可能导致缓冲区溢出。在本例中，一个vector整数容器从src复制到dest通过copy函数实现。由于copy不会做任何事情去扩展vector，当拷贝第一个元素时程序会溢出。

```
#include <algorithm>
#include <vector>

void f(const std::vector<int> &src) {
  std::vector<int> dest;
  std::copy(src.begin(), src.end(), dest.begin());
  // ...
}
```

合规代码

在本例中，正确使用copy的方式是确保目的容器可以容纳要复制的数据，本例中在复制函数执行之前，已经扩大了容器的容量。

```
#include <algorithm>
#include <vector>
void f(const std::vector<int> &src) {
  // Initialize dest with src.size() default-inserted elements
  std::vector<int> dest(src.size());
  std::copy(src.begin(), src.end(), dest.begin());
  // ...
```

```
}
```

合规代码（每个元素增长）

在本例中，另外一种可选方法是应用back_insert_iterator 作为目的参数。该迭代器扩展目的容器为每一次复制扩展元素，确保了容器有足够的大小来容纳每个元素。

```
#include <algorithm>
#include <iterator>
#include <vector>

void f(const std::vector<int> &src) {
  std::vector<int> dest;
  std::copy(src.begin(), src.end(), std::back_inserter(dest));
  // ...
}
```

合规代码（拷贝构造）

在本例中，简洁的方式是通过src拷贝构造出dest容器。

```
#include <vector>

void f(const std::vector<int> &src) {
  std::vector<int> dest(src);
  // ...
}
```

不合规代码

在本例中，std::fill_n()用0x42填充一个缓冲区10个实例大小。然而，容器并没有为缓冲区分配大小，导致了缓冲区溢出。

```
#include <algorithm>
#include <vector>

void f() {
  std::vector<int> v;
  std::fill_n(v.begin(), 10, 0x42);
}
```

合规代码（初始化足够的容量）

在本例中，执行fill_n操作之前，需要确保vector容器容量是足够的。

```
#include <algorithm>
#include <vector>

void f() {
  std::vector<int> v(10);
  std::fill_n(v.begin(), 10, 0x42);
}
```

然而该解决方法是低效的。构造函数中构造了10个int元素，然后用fill_n操作替代，意味着每个元素被容器初始化了2次。

合规代码（初始化填充）

在本例中，初始化容器v使用10个元素用0x42填写。

```
#include <algorithm>
#include <vector>

void f() {
  std::vector<int> v(10, 0x42);
}
```

参考文献

-CTR52-CPP. Guarantee that library functions do not overflow

CTR53-CPP. 使用有效的迭代器范围

作者： 李道春10073354 刘涛10054087 评审人：傅锦华10108953

当遍历迭代器中的所有元素时，迭代器必须要在有效的范围内。一个迭代器的的有效范围是一对分别引用第一个和最后一个元素的迭代器。
一个迭代器的范围有如下特性：

- 迭代器都引用相同的容器
- 代表开始的迭代器先于代表结束的迭代器

- 迭代器都是有效的

一个空的迭代器范围也被认为是有效的。
使用一对无效的迭代器遍历导致无法预测的结果。

不合规代码

在本例中，两个针对同一个容器的迭代器并没有限制范围，但是std::for_each()函数的参数顺序是最后元素迭代器先于第一个元素迭代器。在每一个迭代器的内部循环中，std::for_each()比较第一个迭代器（增加之后）用第二个迭代器；只要它们不等，它会继续对第一个迭代器加1。持续的加1使得最后元素迭代器导致不可预测的结果。

```
#include <algorithm>
#include <iostream>
#include <vector>

void f(const std::vector<int> &c) {
  std::for_each(c.end(), c.begin(), [](int i) { std::cout << i; });
}
```

合规代码

在本例中，迭代器以正确的顺序传递正确的值给std::for_each()函数。

```
#include <algorithm>
#include <iostream>
#include <vector>

void f(const std::vector<int> &c) {
  std::for_each(c.begin(), c.end(), [](int i) { std::cout << i; });
}
```

不合规代码

在本例中，从不同容器生成的迭代器被传递到同一个迭代器范围中。

```
#include <algorithm>
#include <iostream>
#include <vector>

void f(const std::vector<int> &c) {
```

```
  std::vector<int>::const_iterator e;
  std::for_each(c.begin(), e, [](int i) { std::cout << i; });
}
```

合规代码

在本例中，正确的迭代器由end函数产生。

```
#include <algorithm>
#include <iostream>
#include <vector>

void f(const std::vector<int> &c) {
  std::for_each(c.begin(), c.end(), [](int i) { std::cout << i; });
}
```

参考文献

- CTR53-CPP. Use valid iterator ranges

CTR54-CPP. 不要减去来自不同容器的迭代器

作者： 李道春10073354 刘涛10054087 评审人：傅锦华10108953

当两个指针相减时，都需要指向同一个数组对象的元素。该结果不同于两个数组对象下标的相减。类似的，两个迭代器相减，这两个迭代器必须指向相同容器的对象。
如果两个无关的迭代器相减，该操作导致无法预料的后果。

不合规代码

在本例中，尝试确定指针test是否在范围[r, r+n]中。然而，当test不指向给定的范围，在该例子中会导致无法预测的结果。

```
#include <cstddef>
#include <iostream>

template <typename Ty>
bool in_range(const Ty *test, const Ty *r, size_t n) {
  return 0 <= (test - r) && (test - r) <= (std::ptrdiff_t)n;
}
```

```
void f() {
  double foo[10];
  double *x = &foo[0];
  double bar;
  std::cout << std::boolalpha << in_range(&bar, x, 10);
}
```

不合规代码

在本例中，比较指向不同容器的指针会导致无法预测的结果。

```
#include <iostream>
#include <iterator>
#include <vector>

template <typename RandIter>
bool in_range_impl(RandIter test, RandIter r_begin, RandIter r_end, std::random_acce
ss_iterator_tag) {
  return test >= r_begin && test <= r_end;
}

template <typename Iter>
bool in_range(Iter test, Iter r_begin, Iter r_end) {
  typename std::iterator_traits<Iter>::iterator_category cat;
  return in_range_impl(test, r_begin, r_end, cat);
}

void f() {
  std::vector<double> foo(10);
  std::vector<double> bar(1);
  std::cout << std::boolalpha << in_range(bar.begin(), foo.begin(), foo.end());
}
```

合规代码

在本例中，展示了一个可移植的但可能是低效的in_range实现。比较test跟每一个可能的地址[r, n].一个即满足高效又满足可移植的方案还未找到。

```
#include <iostream>
```

```
template <typename Ty>
bool in_range(const Ty *test, const Ty *r, size_t n) {
  auto *cur = reinterpret_cast<const unsigned char *>(r);
  auto *end = reinterpret_cast<const unsigned char *>(r + n);
  auto *testPtr = reinterpret_cast<const unsigned char *>(test);

  for (; cur != end; ++cur) {
    if (cur == testPtr) {
      return true;
    }
  }
  return false;
}

void f() {
  double foo[10];
  double *x = &foo[0];
  double bar;
  std::cout << std::boolalpha << in_range(&bar, x, 10);
}
```

参考文献

- CTR54-CPP. Do not subtract iterators that do not refer to the same container

CTR55-CPP. 在一个可能导致溢出的迭代器上不要使用添加操作符

作者： 李道春10073354 刘涛10054087 评审人：傅锦华10108953

一个指针可以通过一个添加或相减整数类型的表达式，得到一个指针类型的值。此结果指针可能是一个无效的容器成员，其执行结果无法预测。
由于迭代器是泛型指针，当使用随机访问迭代器时，添加操作符也有相同的约束。
不允许对一个随机访问的迭代器指针进行整数值的相加，或者相减，可能会导致结果超过了容器的边界。

不合规代码

在本例中，一个从vector生成的随机访问的迭代器用在加法表达式中，但是结果值可能会超过容器的边界。

```
#include <iostream>
#include <vector>

void f(const std::vector<int> &c) {
  for (auto i = c.begin(), e = i + 20; i != e; ++i) {
    std::cout << *i << std::endl;
  }
}
```

合规代码

在本例中，假定程序员的意图是给容器处理前20个。替代假定元素超过20个元素，容器的大小用于确定累加的上限。

```
#include <algorithm>
#include <vector>

void f(const std::vector<int> &c) {
  const std::vector<int>::size_type maxSize = 20;
  for (auto i = c.begin(), e = i + std::min(maxSize, c.size()); i != e; ++i) {
    // ...
  }
}
```

参考文献

- CTR55-CPP. Do not use an additive operator on an iterator if the result would overflow

CTR56-CPP. 不要在一个多态对象上进行指针运算

作者： 李道春10073354 刘涛10054087 评审人：傅锦华10108953

指针运算并没有计算多态对象的大小，如果尝试在一个多态对象上执行指针的算术运算，结果无法预测。
不要对多态对象的指针进行算术运算，包括数组相减等。

下面的代码假定有如下的静态变量和类定义

```
int globI;
```

```
double globD;

struct S {
  int i;

  S() : i(globI++) {}
};

struct T : S {
  double d;

  T() : S(), d(globD++) {}
};
```

不合规代码

在本例中，f() 接收对象数组S作为第一个参数。然而，main函数传递了T对象数组作为f的实参，结果导致在进行指针的算术运算，产生不可预测的行为。

```
#include <iostream>

// ... definitions for S, T, globI, globD ...

void f(const S *someSes, std::size_t count) {
  for (const S *end = someSes + count; someSes != end; ++someSes) {
    std::cout << someSes->i << std::endl;
  }
}

int main() {
  T test[5];
  f(test, 5);
}
```

不合规代码(数组下标)

在本例中，for循环使用了数组下标。自从数组下标通过指针进行计算，该代码同样会导致无法预测的结果。

```
#include <iostream>
```

```
// ... definitions for S, T, globI, globD ...

void f(const S *someSes, std::size_t count) {
  for (std::size_t i = 0; i < count; ++i) {
    std::cout << someSes[i].i << std::endl;
  }
}

int main() {
  T test[5];
  f(test, 5);
}
```

合规代码(数组)

替代对象数组，一个指针数组解决了对象不同大小的问题，例如如下合规代码代码。

```
#include <iostream>

// ... definitions for S, T, globI, globD ...

void f(const S * const *someSes, std::size_t count) {
  for (const S * const *end = someSes + count; someSes != end; ++someSes) {
    std::cout << (*someSes)->i << std::endl;
  }
}

int main() {
  S *test[] = {new T, new T, new T, new T, new T};
  f(test, 5);
  for (auto v : test) {
    delete v;
  }
}
```

合规代码

另外一种方法是使用标准模板库替代数组并且f()接收迭代器作为参数，例如如下的解决方案。

```
#include <iostream>
```

```
#include <vector>

// ... definitions for S, T, globI, globD ...
template <typename Iter>
void f(Iter i, Iter e) {
  for (; i != e; ++i) {
    std::cout << (*i)->i << std::endl;
  }
}

int main() {
  std::vector<S *> test{new T, new T, new T, new T, new T};
  f(test.cbegin(), test.cend());
  for (auto v : test) {
    delete v;
  }
}
```

参考文献

-CTR56-CPP. Do not use pointer arithmetic on polymorphic objects

CTR57-CPP. 提供有效的排序谓词

作者： 李道春10073354 刘涛10054087 评审人：傅锦华10108953

容器关联时有一个严格的小于运算需求，用在健值的谓词比较上。一个严格的小于运算有如下的属性：
1、for all x: x < x == false (反自反性)
2、for all x, y: if x < y then !(y < x) (不对称性)
3、r all x, y, z: if x < y && y < z then x < z (传递性)
为一个关联容器（例如sets、maps、multisets和multimaps等）或作为排序算法的比较规则而提供的无效排序谓词会导致不规则行为或者导致无限循环。当关联容器或者模板库泛化算法需要一个排序谓词时，此谓词必须满足严格的小于运算要求。

不合规代码

在本例中，一个从set对象使用一个比较器进行创建，它并不满足严格的小于运算要求。特别地，它会返回失败当比较相同值。结果，equal_range遍历结果无法预料。

```
#include <functional>
#include <iostream>
#include <set>

void f() {
  std::set<int, std::less_equal<int>> s{5, 10, 20};
  for (auto r = s.equal_range(10); r.first != r.second; ++r.first) {
    std::cout << *r.first << std::endl;
  }
}
```

合规代码

在本例中，使用一个默认的比较谓词替代一个无效的谓词。

```
#include <iostream>
#include <set>

void f() {
  std::set<int> s{5, 10, 20};
  for (auto r = s.equal_range(10); r.first != r.second; ++r.first) {
    std::cout << *r.first << std::endl;
  }
}
```

不合规代码

在本例中，存储在set里的对象有一个重载小于号，允许对象可以用less进行比较。然而，比较运算并不满足严格的小于运算要求。特别地，两个sets，x 和 y, 他们的values值都是1，但是有不同的值导致在不同情况下comp(x, y) 和comp(y,x)都是false，不满足“不对称性”要求。

```
#include <iostream>
#include <set>

class S {
  int i, j;

public:
  S(int i, int j) : i(i), j(j) {}
```

```
  friend bool operator<(const S &lhs, const S &rhs) {
    return lhs.i < rhs.i && lhs.j < rhs.j;
  }

  friend std::ostream &operator<<(std::ostream &os, const S& o) {
    os << "i: " << o.i << ", j: " << o.j;
    return os;
  }
};

void f() {
  std::set<S> t{S(1, 1), S(1, 2), S(2, 1)};
  for (auto v : t) {
    std::cout << v << std::endl;
  }
```

合规代码

在本例中，使用tie来实现严格的小于运算要求。

```
#include <iostream>
#include <set>
#include <tuple>

class S {
  int i, j;

public:
  S(int i, int j) : i(i), j(j) {}

  friend bool operator<(const S &lhs, const S &rhs) {
    return std::tie(lhs.i, lhs.j) < std::tie(rhs.i, rhs.j);
  }

  friend std::ostream &operator<<(std::ostream &os, const S& o) {
    os << "i: " << o.i << ", j: " << o.j;
    return os;
  }
};

void f() {
  std::set<S> t{S(1, 1), S(1, 2), S(2, 1)};
```

```
  for (auto v : t) {
    std::cout << v << std::endl;
  }
}
```

参考文献

- CTR57-CPP. Provide a valid ordering predicate

CTR58-CPP. 谓词函数对象不应该可变

作者： 李道春10073354 刘涛10054087 评审人：傅锦华10108953

C模板库中实现了许多通用的接收谓词对象的算法。C标准中描述如下：

[Note: Unless otherwise specified, algorithms that take function objects as arguments are permitted to copy those function objects freely. Programmers for whom object identity is important should consider using a wrapper class that points to a noncopied implementation object such as reference_wrapper, or some equivalent solution. — end note]
由该条目是实现定义算法是否会拷贝一个谓词函数对象，这类对象必须要：
1）实现一个函数调用运算符，其不可改变和函数对象标示符相关的状态，例如非静态数据成员，或者
2）将谓词函数对象封装在一个std::reference_wrapper (或等效解决方案)
将函数调用运算符标记为const比较好，但是有可能失效，因为带有mutable存储类关键字的对象成员依然可能会被一个const成员函数所修改。

不合规代码(函子)

在本例中，当试图删除第三个容器中的元素时，使用了一个仅在第三次调用时返回true的谓词。

```
#include <algorithm>
#include <functional>
#include <iostream>
#include <iterator>
#include <vector>

class MutablePredicate : public std::unary_function<int, bool> {
  size_t timesCalled;
```

```
public:
  MutablePredicate() : timesCalled(0) {}

  bool operator()(const int &) {
    return ++timesCalled == 3;
  }
};

template <typename Iter>
void print_container(Iter b, Iter e) {
  std::cout << "Contains: ";
  std::copy(b, e, std::ostream_iterator<decltype(*b)>(std::cout, " "));
  std::cout << std::endl;
}

void f() {
  std::vector<int> v{0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
  print_container(v.begin(), v.end());

  v.erase(std::remove_if(v.begin(), v.end(), MutablePredicate()), v.end());
  print_container(v.begin(), v.end());
}
```

合规代码(std::reference_wrapper)

在本例中，使用std::reference_wrapper封装了谓词对象。确保了所有复制的封装对象都指向相同的谓词对象。

```
#include <algorithm>
#include <functional>
#include <iostream>
#include <iterator>
#include <vector>

class MutablePredicate : public std::unary_function<int, bool> {
  size_t timesCalled;
public:
  MutablePredicate() : timesCalled(0) {}

  bool operator()(const int &) {
    return ++timesCalled == 3;
  }
```

```
};

template <typename Iter>
void print_container(Iter b, Iter e) {
  std::cout << "Contains: ";
  std::copy(b, e, std::ostream_iterator<decltype(*b)>(std::cout, " "));
  std::cout << std::endl;
}

void f() {
  std::vector<int> v{0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
  print_container(v.begin(), v.end());

  MutablePredicate mp;
  v.erase(std::remove_if(v.begin(), v.end(), std::ref(mp)), v.end());
  print_container(v.begin(), v.end());
}
```

合规代码(迭代器算法)

在本例中，删除容器中一个特定的元素并不一定要用谓词函数，可以使用std::vector::erase函数替代，例如下面的代码示例。

```
#include <algorithm>
#include <iostream>
#include <iterator>
#include <vector>

template <typename Iter>
void print_container(Iter B, Iter E) {
  std::cout << "Contains: ";
  std::copy(B, E, std::ostream_iterator<decltype(*B)>(std::cout, " "));
  std::cout << std::endl;
}

void f() {
  std::vector<int> v{0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
  print_container(v.begin(), v.end());
  v.erase(v.begin() + 3);
  print_container(v.begin(), v.end());
}
```

参考文献

- CTR58-CPP. Predicate function objects should not be mutable

Rule 05. 字符和字符串

STR50-CPP. 保证字符串有足够的存储空间存放字符数据和结束符标识

作者： 李道春10073354 刘涛10054087 评审人： 赵庆轩10146152

拷贝数据到一个空间不够大的缓冲区将导致缓冲区溢出。使用字符串的时候，经常发生缓冲区溢出。为了避免这样的错误，可以通过截断的方式限制拷贝。更好的方法是，确保目标端有足够大的空间容纳待拷贝的数据。C语言风格字符串需要一个结束标识来表明已经到了结束串末尾。

不合规代码

因为输入未绑定，下面代码将导致缓冲区溢出。

```
#include <iostream>

void f() {
  char buf[12];
  std::cin >> buf;
}
```

合规代码

要确保数据不被截断，并且防止缓冲区溢出，最佳的方法是使用std::string替代绑定数组的方法。正确的范例如下：

```
#include <iostream>
#include <string>

void f() {
  std::string input;
```

```
  std::string stringOne;
  std::cin >> stringOne;
}
```

不合规代码

在本例中，非格式化输入函数std::basic_istream::read()读入一个未格式化的32个字符的数组，该数组来自指定文件。但是，read()函数不保证字符串用空字符结尾。如果数组没有包含字符串结束标识，接下来调用std::string将产生不可预期的结果。

```
#include <fstream>
#include <string>

void f(std::istream &in) {
  char buffer[32];
  try {
    in.read(buffer, sizeof(buffer));
  } catch (std::ios_base::failure &e) {
    // Handle error
  }

  std::string str(buffer);
  // ...
}
```

合规代码

在本例中，假设文件输入最多32字节。根据输入流中的字符数量，这段代码构造了一个std::string对象，而不是插入一个空结束标识。如果输入字节长度不确定，最好使用std::basic_istream::readsome()，或根据需要使用格式化的输入函数。

```
#include <fstream>
#include <string>

void f(std::istream &in) {
  char buffer[32];
  try {
    in.read(buffer, sizeof(buffer));
  } catch (std::ios_base::failure &e) {
    // Handle error
  }
```

```
  std::string str(buffer, in.gcount());
  // ...
}
```

参考文献

- STR50-CPP. Guarantee that storage for strings has sufficient space for character data and the null terminator

STR51-CPP. 不要试图用空指针创建std::string

作者： 李道春10073354 刘涛10054087 评审人： 赵庆轩10146152

不合规代码

在本例中，std::tring对象产生于std::getenv()的调用。然而，因为std::getenv()失败时会返回空指针，当环境变量不存在或发生其他错误时，这段代码会导致未知结果。

```
#include <cstdlib>
#include <string>

void f() {
  std::string tmp(std::getenv("TMP"));
  if (!tmp.empty()) {
    // ...
  }
}
```

合规代码

在本例中，构造std::string对象之前，调用std::getenv()后会检查空指针。

```
#include <cstdlib>
#include <string>

void f() {
  const char *tmpPtrVal = std::getenv("TMP");
  std::string tmp(tmpPtrVal ? tmpPtrVal : "");
  if (!tmp.empty()) {
    // ...
```

```
  }
}
```

参考文献

- STR51-CPP. Do not attempt to create a std::string from a null pointer

STR52-CPP. 使用有效的引用、指针和迭代器来引用basic_string中的元素

作者： 李道春10073354 刘涛10054087 评审人： 赵庆轩10146152

因为 std::basic_string 是一种字符容器，因此这个规则是规则 的一个特例。作为容器，std::basic_string支持像标准模板库的其他容器一样使用迭代器。然而，std::basic_string模板类有异常失效的语义。C++标准中， [string.require字符串需求]，第5节规定如下：
对basic_string对象的如下用法可能会导致对basic_string序列元素的引用、指针或迭代器变成无效的：

1. 使用以非const的basic_string对象作为参数的标准库函数。
2. 调用除操作符[], at, front, back, begin, rbegin, end, 和 rend外的非const成员函数。
   使用非const的std::basic_string模板类的标准库函数例子有：std::swap(), ::operator>>(basic_istream &, string &), 和 std::getline()。
   不要使用一个无效的引用、指针或迭代器，因为这样做会导致行为不可预测。

不合规代码

```
#include <string>

void f(const std::string &input) {
  std::string email;

  // Copy input into email converting ";" to " "
  std::string::iterator loc = email.begin();
  for (auto i = input.begin(), e = input.end(); i != e; ++i, ++loc) {
    email.insert(loc, *i != ';' ? *i : ' ');
  }
}
```

上面的不合规代码将input复制到std::string，用空格字符替换分号字符（;）。这个例子不

符合规范是因为在第一次调用insert()后迭代器loc会失效。后续调用insert()的行为是未定义的。

合规代码（std::string::insert()）

```
#include <string>

void f(const std::string &input) {
  std::string email;

  // Copy input into email converting ";" to " "
  std::string::iterator loc = email.begin();
  for (auto i = input.begin(), e = input.end(); i != e; ++i, ++loc) {
    loc = email.insert(loc, *i != ';' ? *i : ' ');
  }
}
```

在这个合规代码中，迭代器loc的值由于每次调用insert()都会被更新，因此不会访问到无效迭代器。更新后的迭代器会在循环最后递增。

合规代码（std::replace()）

```
#include <algorithm>
#include <string>

void f(const std::string &input) {
  std::string email{input};
  std::replace(email.begin(), email.end(), ';', ' ');
}
```

这个合规代码使用标准算法进行替换操作。如果可能，使用一个通用的算法要好过你自己的解决方案。

不合规代码

```
#include <iostream>
#include <string>

extern void g(const char *);
```

```
void f(std::string &exampleString) {
  const char *data = exampleString.data();
  // ...
  exampleString.replace(0, 2, "bb");
  // ...
  g(data);
}
```

在这个不合规代码中，调用replace()后，data会变成无效的，所以在g()中使用data会导致行为不可预测。

合规代码

```
#include <iostream>
#include <string>

extern void g(const char *);

void f(std::string &exampleString) {
  // ...
  exampleString.replace(0, 2, "bb");
  // ...
  g(exampleString.data());
}
```

在这个合规代码中，replace()的修改完成后，指向exampleString的内部缓冲区才会生成。

参考文献

- STR52-CPP. Use valid references, pointers, and iterators to reference elements of a basic_string

STR53-CPP. 元素访问范围检查

作者： 李道春10073354 刘涛10054087 评审人： 赵庆轩10146152

Std::string索引运算符const_reference operator [] (size_type) const 和 reference operator[] (size_type) 返回存储在指定位置pos的字符。当 pos >= size()，会返回具有值charT()的charT类型的对象引用。索引运算符是未检查的（不会抛出范围错误的异常），并且企图修改结果中超出范围的对象会导致未定义的行为。

同样，std::string::back()和 std::string::front()函数也不会做检查，因为它们定义为调用相应的operator，而operator不会抛出异常。

不要给std::string::operator[] ()传递一个超出范围的值作为参数。同样，不要在一个空字符串上调用std::string::back() 或 std::string::front()。这条规则是规则 CTR50-CPP.保证容器索引和迭代器在有效范围内 的特例。

不合规代码

```
#include <string>

extern std::size_t get_index();

void f() {
  std::string s("01234567");
  s[get_index()] = '1';
}
```

在这个不合规代码中，调用get_index()的返回值可能会大于存储的字符串中元素的数量，从而导致未定义的行为。

合规代码（try/catch）(C++11支持)

```
#include <stdexcept>
#include <string>
extern std::size_t get_index();

void f() {
  std::string s("01234567");
  try {
    s.at(get_index()) = '1';
  } catch (std::out_of_range &) {
    // Handle error
  }
}
```

这个合规代码采用std::basic_string::at()函数，它和operator[]具有类似行为，但是如果pos >= size()，会抛出std::out_of_range异常。

合规代码（采用范围检查）

```
#include <string>

extern std::size_t get_index();

void f() {
  std::string s("01234567");
  std::size_t i = get_index();
  if (i < s.length()) {
    s[i] = '1';
  } else {
    // Handle error
  }
}
```

这个合规代码在调用operator[] ()之前，会检查get_index()返回值是否在有效范围内。

不合规代码

```
#include <string>
#include <locale>

void capitalize(std::string &s) {
  std::locale loc;
  s.front() = std::use_facet<std::ctype<char>>(loc).toupper(s.front());
}
```

这个不合规代码试图用等效的大写字符替换字符串中的首字符。然而，如果给定的字符串为空，会导致行为不可预测。

合规代码

```
#include <string>
#include <locale>

void capitalize(std::string &s) {
  if (s.empty()) {
    return;
  }

  std::locale loc;
  s.front() = std::use_facet<std::ctype<char>>(loc).toupper(s.front());
```

```
}
```

在这个合规代码中，仅当字符串不为空时才执行std::string::front()调用。

参考文献

- STR53-CPP. Range check element access

Rule 06. 内存管理

MEM50-CPP. 禁止访问已释放的内存

作者： 刘光聪10209986 评审人： 曾亮亮10091330

指向已经释放的内存的指针称为悬挂指针。访问悬挂指针，其程序行为是未定义的。

不规范代码(new和delete)

在该不规范的代码示例中， `s` 在它的内存被释放后再解引用。

```
#include <new>

struct S {
  void f();
};

void g() noexcept(false) {
  S *s = new S;
  // ...
  delete s;
  // ...
  s->f();
}
```

合规代码(new和delete)

在合规代码示例中，动态分配的内存直到不再使用时才被释放。

```
#include <new>
struct S {
  void f();
};
void g() noexcept(false) {
  S *s = new S;
  // ...
  s->f();
  delete s;
}
```

规范代码（自动存储期）

尽最大可能，用自动存储期代替动态存储器。因为变量 s 在函数 g() 的作用域外不需要存活，在此合规代码示例中，采用自动存储器限制函数 g() 作用域内变量 s 的生存期。

```
struct S {
  void f();
};

void g() {
  S s;
  // ...
  s.f();
}
```

不合规代码(std::unique_ptr)

在此不合规代码示例中，由 buff 动态分配的内存，当对象在析构函数中隐式释放之后又被再次访问。

```
#include <iostream>
#include <memory>
#include <cstring>

int main(int argc, const char *argv[]) {
  const char *s = "";
  if (argc > 1) {
    enum { BufferSize = 32 };
    try {
      std::unique_ptr<char[]> buff(new char[BufferSize]);
```

```
      std::memset(buff.get(), 0, BufferSize);
      // ...
      s = std::strncpy(buff.get(), argv[1], BufferSize - 1);
    } catch (std::bad_alloc &) {
      // Handle error
    }
  }

  std::cout << s << std::endl;
}
```

合规代码(std::unique_ptr)

在合规代码中， `buff` 对象的生命周期延长，使得内存可用。

```
#include <iostream>
#include <memory>
#include <cstring>

int main(int argc, const char *argv[]) {
  std::unique_ptr<char[]> buff;
  const char *s = "";

  if (argc > 1) {
    enum { BufferSize = 32 };
    try {
      buff.reset(new char[BufferSize]);
      std::memset(buff.get(), 0, BufferSize);
      // ...
      s = std::strncpy(buff.get(), argv[1], BufferSize - 1);
    } catch (std::bad_alloc &) {
      // Handle error
    }
  }

  std::cout << s << std::endl;
}
```

合规代码

在该合规代码示例中，使用 `std::string` 类型的自动存储期变量来代替

`std::unique_ptr<char[]>` ，既降低了复杂性，又提高了安全性。

```
#include <iostream>
#include <string>

int main(int argc, const char *argv[]) {
  std::string str;

  if (argc > 1) {
    str = argv[1];
  }

  std::cout << str << std::endl;
}
```

不合规代码(std::string::c_str())

在该不合规代码示例中， `std::string::c_str()` 的调用发生在一个临时的 `std::string` 对象上。一旦该临时的 `std::string` 对象在赋值表达式的末尾被销毁，将会使指针指向已释放的内存，从而导致访问该指针的元素时出现未定义的行为。

```
#include <string>

std::string str_func();
void display_string(const char *);

void f() {
  const char *str = str_func().c_str();
  display_string(str);  /* Undefined behavior */
}
```

合规代码(std::string::c_str())

在该合规代码中，函数 `str_func` 返回的字符串的本地副本，可以确保在调用 `display_string()` 时字符串 `str` 仍然有效。

```
#include <string>

std::string str_func();
void display_string(const char *s);
```

```
void f() {
  std::string str = str_func();
  const char *cstr = str.c_str();
  display_string(cstr);  /* ok */
}
```

不合规代码

在该不合规代码示例中，尝试通过对 operator new() 的调用来分配零字节的内存。如果此请求成功，则需要 operator new() 返回非空指针。

```
#include <new>
void f() noexcept(false) {
  unsigned char *ptr = static_cast<unsigned char *>(::operator new(0));
  *ptr = 0;
  // ...
  ::operator delete(ptr);
}
```

合规代码

在该合规代码中，程序员打算分配一个 unsigned char 对象，则使用 new 而不是直接调用 operator new() 。

```
void f() noexcept(false) {
  unsigned char *ptr = new unsigned char;
  *ptr = 0;
  // ...
  delete ptr;
}
```

合规代码

如果程序员打算分配0字节的内存（可能获得一个唯一的指针值，它不能被程序中的任何其他指针重复使用，直到它被正确释放），那么推荐的解决方案是将 ptr 声明为 void* ，而不是通过一致的实现来解引用。

```
#include <new>

void f() noexcept(false) {
```

```
  void *ptr = ::operator new(0);
  // ...
  ::operator delete(ptr);
}
```

MEM51-CPP. 正确释放动态分配的资源

作者： 刘光聪10209986 评审人： 曾亮亮10091330

C程序言提供了几种分配内存的方法，比如 `std::malloc()` ， `std::calloc()` 和 `std::realloc()` ，也可以由C程序使用。然而，C语言只定义了一种释放内存的方法: `std::free()` 。

`C 程序语言增加了分配内存的其他方法，比如运算符 new、 new[] 、 定位放置 new(placement new) 和分配器对象( std::allocator )。不像C语言， C++ 语言提供了多种方式来释放动态分配的内存，比如运算符 delete、delete[]()`和分配器对象提供的释放函数。

分配器	释放器
global operator new()/new	global operator delete()/delete
global operator new/new[]	global operator delete/delete[]
class-specific operator new()/new	class-specific operator delete()/delete
class-specific operator new/new[]	class-specific operator delete/delete[]
placement operator new()	N/A
allocator<T>::allocate()	allocator<T>::deallocate()
std::malloc(), std::calloc(), std::realloc()	std::free()
std::get_temporary_buffer()	std::return_temporary_buffer()

不合规代码(placement new())

在这个不合规代码示例中，局部变量 `s1` 作为表达式传递给 `placement new` 运算符。然后将返回的指针传递给 `::operator delete()` ，从而导致未定义的行为，因为 `::operator delete()` 试图释放由 `::operator new()` 返回的内存。

```
#include <iostream>
```

```
struct S {
  S() { std::cout << "S::S()" << std::endl; }
  ~S() { std::cout << "S::~S()" << std::endl; }
};

void f() {
  S s1;
  S *s2 = new (&s1) S;

  // ...

  delete s2;
}
```

合规代码

在该合规代码中，删除了对 `::operator delete()` 的调用， `s1` 对象因生命周期终止而被销毁。

```
#include <iostream>

struct S {
  S() { std::cout << "S::S()" << std::endl; }
  ~S() { std::cout << "S::~S()" << std::endl; }
};

void f() {
  S s1;
  S *s2 = new (&s1) S;

  // ...
}
```

不合规代码（未初始化的delete）

在该不合规代码示例中，在同一个 `try` 块中尝试两次分配，如果任一个失败， `catch` 处理程序会尝试释放已分配的资源。但是，当 `i1` 分配内存的失败，导致未初始化的 `i2` 被非法 `delete` ，其程序行为是未定义的。

```
#include <new>
```

```
void f() {
  int *i1, *i2;
  try {
    i1 = new int;
    i2 = new int;
  } catch (std::bad_alloc &) {
    delete i1;
    delete i2;
  }
}
```

合规代码

在该合规代码中，将两个指针的值初始化为 `nullptr` 。

```
#include <new>

void f() {
  int *i1 = nullptr, *i2 = nullptr;
  try {
    i1 = new int;
    i2 = new int;
  } catch (std::bad_alloc &) {
    delete i1;
    delete i2;
  }
}
```

不合规代码（双重释放）

在该不合规代码示例中，类 `C` 持有 `P*` ，随后被类的析构函数释放。尽管存在一个用户声明的析构函数， `C` 将合成一个隐式的拷贝构造函数，这个拷贝构造函数将复制存储在 `p` 中的指针值，导致双重释放：第一次释放发生在 `g()` 退出时，第二个释放发生在 `h()` 退出时。

```
struct P {};

class C {
  P *p;

public:
```

```
  C(P *p) : p(p) {}
  ~C() { delete p; }

  void f() {}
};

void g(C c) {
  c.f();
}

void h() {
  P *p = new P;
  C c(p);
  g(c);
}
```

合规代码（双重释放）

在该合规代码中，类 `C` 的拷贝构造函数和拷贝赋值运算符被显式地删除。由于使用了删除的拷贝构造函数，这种删除将导致像前述不合规代码示例中， `g()` 的实现导致编译失败。 因此， `g()` 被修改以通过引用传递参数，避免双重释放。

```
struct P {};

class C {
  P *p;

public:
  C(P *p) : p(p) {}
  C(const C&) = delete;
  ~C() { delete p; }

  void operator=(const C&) = delete;

  void f() {}
};

void g(C &c) {
  c.f();
}

void h() {
```

```
  P *p = new P;
  C c(p);
  g(c);
}
```

不合规代码(array new[])

在该不合规代码示例中，数组通过 `new[]` 分配内存，但是通过调用 `delete` 而不是 `delete[]` 来释放内存，导致未定义的行为。

```
void f() {
  int *array = new int[10];
  // ...
  delete array;
}
```

合规代码(array new[])

在该合规代码中，通过调用 `delete[]` 而不是 `delete` 来释放内存，遵循正确的内存分配和释放函数的配对来固定代码。

```
void f() {
  int *array = new int[10];
  // ...
  delete[] array;
}
```

不合规代码(malloc())

在该不合规代码示例中， `malloc()` 的调用和 `delete` 的调用混合

```
#include <cstdlib>
void f() {
  int *i = static_cast<int *>(std::malloc(sizeof(int)));
  // ...
  delete i;
}
```

实现细节

`::operator new()` 的一些实现导致调用 `std::malloc()` 。在这样的实现上， `::operator delete()` 函数需要调用 `std::free()` 来释放指针，而不合规的代码示例将以良好定义的方式运行。但是，这是一个实现细节，不应依赖，即没有义务使用底层C内存管理函数来实现 `C++` 的内存管理运算符。

合规代码(malloc())

在该合规代码中，由 `std::malloc()` 分配的指针通过调用 `std::free()` 而不是 `delete` 来释放。

```
#include <cstdlib>

void f() {
  int *i = static_cast<int *>(std::malloc(sizeof(int)));
  // ...
  std::free(i);
}
```

不合规代码(new)

在该不合规代码示例中， `std::free()` 被调用来释放由 `new` 分配的内存。使用不正确的释放函数引起的未定义行为，其常见的副作用将不会为通过 `std::free()` 释放的对象调用析构函数。

```
#include <cstdlib>

struct S {
  ~S();
};

void f() {
  S *s = new S();
  // ...
  std::free(s);
}
```

合规代码(new)

在该合规代码中，由 `new` 分配的指针通过调用 `delete` 而不是 `std::free()` 来释放。

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

void f() {
  S *s = new S();
  // ...
  delete s;
}
```

不合规代码(Class new)

在该不合规代码示例中，全局 new 运算符被具体类的运算符 new() 的实现覆盖。 当调用 new 时，具体类的覆写被选择，因此 S::operator new() 被调用。但是，因为对象通过全局 delete 运算符销毁，所以调用全局的 delete() 函数而不是具体类的实现 S::operator delete() ，导致了未定义的行为。

```
#include <cstdlib>
#include <new>

struct S {
  static void *operator new(std::size_t size) noexcept(true) {
    return std::malloc(size);
  }

  static void operator delete(void *ptr) noexcept(true) {
    std::free(ptr);
  }
};

void f() {
  S *s = new S;
  ::delete s;
}
```

合规代码(Class new)

在该合规代码中，通过局部 delete 调用代替全局 delete 调用导致 S::operator delete() 被调用。

```
#include <cstdlib>
#include <new>

struct S {
  static void *operator new(std::size_t size) noexcept(true) {
    return std::malloc(size);
  }

  static void operator delete(void *ptr) noexcept(true) {
    std::free(ptr);
  }
};

void f() {
  S *s = new S;
  delete s;
}
```

不合规代码(std::unique_ptr)

在该不合规代码示例中， `std::unique_ptr` 被声明为保存对象的指针，但是用对象数组直接初始化。当 `std::unique_ptr` 被销毁时，它的默认删除器调用 `delete` 而不是 `delete[]` ，导致未定义的行为。

```
#include <memory>

struct S {};

void f() {
  std::unique_ptr<S> s{new S[10]};
}
```

合规代码(std::unique_ptr)

在合规代码中， `std::unique_ptr` 被声明为保存对象数组，而不是指向对象的指针。此外， `std::make_unique()` 用于初始化智能指针。

```
#include <memory>

struct S {};
```

```
void f() {
  std::unique_ptr<S[]> s = std::make_unique<S[]>(10);
}
```

如果生成的std::unique_ptr不是正确的类型，使用std::make_unique()而不是直接初始化将发出一个诊断。如果在不规范的代码示例中使用它，结果将是一个不成熟的程序而不是未定义的行为。最好使用std::make_unique()而不是通过其他方式手动初始化。

不合规代码(std::shared_ptr)

与 std::unique_ptr 一样，当 std::shared_ptr 被销毁时，它的默认删除程序调用 delete 而不是 delete[] ，导致未定义的行为。

```
#include <memory>

struct S {};

void f() {
  std::shared_ptr<S> s{new S[10]};
}
```

合规代码(std::unique_ptr)

在此合规代码中，为共享指针类型手动定制了删除程序，确保底层数组被正确的删除。

```
#include <memory>

struct S {};

void f() {
  std::shared_ptr<S> s{new S[10], [](const S *ptr) { delete [] ptr; }};
}
```

MEM52. 检测并捕捉内存分配错误
作者： 刘光聪10209986 评审人： 曾亮亮10091330

默认的内存分配操作 ::operator new(std::size_t) 分配失败后，会抛出一个 std::bad_alloc 异常。因此，你不需要去校验调用 ::operator new(std::size_t) 的结果是否为 nullptr 。还有一种没有异常抛出的形式，例如 ::operator

new(std::size_t, const std::nothrow_t &) 分配失败后，不会抛出异常，但是会把返回值置为 nullptr 。同样的行为也会发生在 operator new[] 上。此外，默认的分配器对象(std::allocator)使用 ::operator new(std::size_t) 去分配，显得更加简单。

```
T *p1 = new T; // Throws std::bad_alloc if allocation fails
T *p2 = new (std::nothrow) T; // Returns nullptr if allocation fails
T *p3 = new T[1]; // Throws std::bad_alloc if the allocation fails
T *p4 = new (std::nothrow) T[1]; // Returns nullptr if the allocation fails
```

与此同时，当传给 new 的 size 参数为负数或者过大时， operator new[] 抛出类型为 std::bad_array_new_length 的错误，该异常是 std::bad_alloc 的子类。当使用不抛异常的形式，那么在使用指针时检查返回值是否为 nullptr 将是非常必要的。

不合规代码

在该不合规代码中，使用 ::operator new[](std::size_t) 创建一个 int 数组，分配的结果并没有校验。函数被标记为 noexcept ，需要调用者确保该函数不会抛出任何异常。因为 ::operator new[](std::size_t) 在分配失败时会抛出异常，它可能会导致程序的异常退出。

```
#include <cstring>

void f(const int *array, std::size_t size) noexcept {
  int *copy = new int[size];
  std::memcpy(copy, array, size * sizeof(*copy));
  // ...
  delete [] copy;
}
```

合规代码（std::nothrow)

当使用 std::nothrow 时， new 操作符返回一个空指针或指向已分配空间的指针。始终测试返回的指针，以确保在引用该指针之前它不为 nullptr 。

```
#include <cstring>
#include <new>

void f(const int *array, std::size_t size) noexcept {
  int *copy = new (std::nothrow) int[size];
  if (!copy) {
```

```
    // Handle error
    return;
  }
  std::memcpy(copy, array, size * sizeof(*copy));
  // ...
  delete [] copy;
}
```

合规代码（std::bad_alloc)

当没有充足的内存分配时，还有一种选择，可以使用不带 `std::nothrow` 的 `::operator new[]` ，替代捕获 `std::bad_alloc` 异常。

```
#include <cstring>
#include <new>

void f(const int *array, std::size_t size) noexcept {
  int *copy;
  try {
    copy = new int[size];
  } catch(std::bad_alloc) {
    // Handle error
    return;
  }
  // At this point, copy has been initialized to allocated memory
  std::memcpy(copy, array, size * sizeof(*copy));
  // ...
  delete [] copy;
}
```

合规代码（noexcept(false)）

如果一个函数像这样设计：希望函数调用者来捕获异常，此时正确的做法是，可以显式的标记该函数可能会返回异常。但是这种做法没有强制要求，因为没有noexcept标识符的函数都默认可以抛出异常。

```
#include <cstring>

void f(const int *array, std::size_t size) noexcept(false) {
  int *copy = new int[size];
  // If the allocation fails, it will throw an exception which the caller
```

```
  // will have to handle.
  std::memcpy(copy, array, size * sizeof(*copy));
  // ...
  delete [] copy;
}
```

MEM53-CPP. 手动管理对象的生命周期时，显式的构造和析构对象

作者： 李永顺10110636 评审人： 曾亮亮10091330

在 C++ 中，对象的动态创建包括两个阶段：第一个阶段负责分配足够的内存来存储对象，第二个阶段负责对象的类型初始化新分配的内存块。相似的，在 C++ 中，动态创建的对象，其析构同样包括两个阶段：第一个阶段是负责根据对象类型销毁对象，第二个阶段是负责回收对象使用的内存。
当手工管理一个对象的生命周期，需要调用构造函数去初始化一个对象的生命周期；相似的，调用析构函数去终止一个对象的生命周期。使用一个在生命周期之外的对象是未定义行为。一个对象的创建可以显式的使用 placement new 来调用，对于对象分配器可以调用 construct() 函数进行创建。一个对象可以使用显式的析构函数调用来销毁，对于对象分配器可以调用 destory() 函数来销毁。

不合规代码

在该不合规代码例子中，一个具有非默认构造的类（因为存在一个用户自定义的构造函数）使用 std::malloc() 创建对象，但是这个对象的构造函数未被调用，随后当通过 s->f() 调用访问时将导致未定义的行为。

```
#include <cstdlib>

struct S {
  S();

  void f();
};

void g() {
  S *s = static_cast<S *>(std::malloc(sizeof(S)));

  s->f();
```

```
  std::free(s);
}
```

合规代码

在该不合规代码例子中，构造函数和析构函数都是显式调用。

```
#include <cstdlib>
#include <new>

struct S {
  S();

  void f();
};

void g() {
  void *ptr = std::malloc(sizeof(S));
  S *s = new (ptr) S;

  s->f();

  s->~S();
  std::free(ptr);
}
```

不合规代码

在该不合规代码示例中，一个自定义容器类使用一个对象分配器获取任意元素的存储。当使用 `copy_elements()` 函数时，假定使用拷贝构造的元素移动到新分配的存储空间，这个例子的错误在于没有显式的调用默认构造函数。如果这样的一个元素调用了它的 `operator[]()` 方法，它会在类型 `T` 上发生未定义行为。

```
#include <memory>

template <typename T, typename Alloc = std::allocator<T>>
class Container {
  T *underlyingStorage;
  size_t numElements;
```

```
  void copy_elements(T *from, T *to, size_t count);

public:
  void reserve(size_t count) {
    if (count > numElements) {
      Alloc alloc;
      T *p = alloc.allocate(count); // Throws on failure
      try {
        copy_elements(underlyingStorage, p, numElements);
      } catch (...) {
        alloc.deallocate(p, count);
        throw;
      }
      underlyingStorage = p;
    }
    numElements = count;
  }

  T &operator[](size_t idx) { return underlyingStorage[idx]; }
  const T &operator[](size_t idx) const { return underlyingStorage[idx]; }
};
```

合规代码

在该合规代码示例中，所有的元素通过显式的调用拷贝构造或者默认构造函数进行了初始化

```
#include <memory>

template <typename T, typename Alloc = std::allocator<T>>
class Container {
  T *underlyingStorage;
  size_t numElements;

  void copy_elements(T *from, T *to, size_t count);

public:
  void reserve(size_t count) {
    if (count > numElements) {
      Alloc alloc;
      T *p = alloc.allocate(count); // Throws on failure
      try {
```

```
        copy_elements(underlyingStorage, p, numElements);
        for (size_t i = numElements; i < count; ++i) {
          alloc.construct(&p[i]);
        }
      } catch (...) {
        alloc.deallocate(p, count);
        throw;
      }
      underlyingStorage = p;
    }
    numElements = count;
  }

  T &operator[](size_t idx) { return underlyingStorage[idx]; }
  const T &operator[](size_t idx) const { return underlyingStorage[idx]; }
};
```

MEM55-CPP. 谨慎替换动态内存管理

作者： 刘光聪10209986 评审人： 曾亮亮10091330

C++ 标准允许用户自定义动态内存分配和释放函数，自定义函数要与*C++*标准定义保持语义相同。

特别是作为库或API时要保持语义一致性，避免异常情况下导致不确定行为（如果仅是用户自己使用，行为都在控制范围之内，则不必强求按照 *C++* 标准执行）。

不合规代码

如下代码中，全局 `operator new(std::size_t)` 在异常情况下，返回空指针，*C++*标准规定异常情况下需要抛出 `std::bad_alloc` 异常，当用户通过捕获异常使用分配内存时，可能由于访问空指针导致程序异常。

```
#include <new>

void *operator new(std::size_t size) {
  extern void *alloc_mem(std::size_t); // Implemented elsewhere; may return nullptr
  return alloc_mem(size);
}

void operator delete(void *ptr) noexcept; // Defined elsewhere
```

```
void operator delete(void *ptr, std::size_t) noexcept; // Defined elsewhere
```

合规代码

如下代码在内存申请失败后抛出 `std::bad_alloc` ，以C++标准要求一致。

```
#include <new>

void *operator new(std::size_t size) {
  extern void *alloc_mem(std::size_t); // Implemented elsewhere; may return nullptr
  if (void *ret = alloc_mem(size)) {
    return ret;
  }
  throw std::bad_alloc();
}

void operator delete(void *ptr) noexcept; // Defined elsewhere
void operator delete(void *ptr, std::size_t) noexcept; // Defined elsewhere
```

参考文献

- MEM55-CPP. Honor replacement dynamic storage management requirements

MEM56-CPP. 不要使用不关联的智能指针存储一个已被接管的指针值

作者： 刘光聪10209986 评审人： 曾亮亮10091330

C++11 丰富了智能指针，使用 `std::unique_ptr` 替换了有缺陷的 `std::auto_ptr` ，另外还新增了 `std:shared_ptr` 和 `std::weak_ptr` 。

`std::unique_ptr` 与其接管的指针指向内存紧绑定，不允许通过赋值构造或者复制进行转移，仅能通过 `std::move` 转移。
`std::shared_ptr` 采用引用计数方式管理指向指针，因此允许多个智能指针共享管理同一内存。还可以通过 `std::shared_ptr` 显示释放内存，因此如果对同一指针，如果使用不相关的 `shared_ptr` 管理，可能导致内存被释放两次异常。

不合规代码

如下代码中，使用两个不相关的智能指针指向同一内存，局部变量 p1,p2 都会释放它们指向的内存。

```
#include <memory>

void f() {
  int *i = new int;
  std::shared_ptr<int> p1(i);
  std::shared_ptr<int> p2(i);
}
```

合规代码

在如下代码中，通过赋值构造另一个已经存在的智能指针 p1 ，使得 p2 与 p1 相关联，共同管理指向的内存。当 p2 被销毁时，仅是计数递减，只有 p1 也被释放时，才会递减为0，释放指向的内存。

```
#include <memory>

void f() {
  std::shared_ptr<int> p1 = std::make_shared<int>();
  std::shared_ptr<int> p2(p1);
}
```

不合规代码

在如下代码中，在通过 dynamic_cast<D *>(poly.get()) 转换时，产生一次智能指针拷贝，而该指针与 poly 不先关，导致 f() 与 g(...) 对同一内存释放两次。

```
#include <memory>

struct B {
  virtual ~B() = default; // Polymorphic object
  // ...
};
struct D : B {};

void g(std::shared_ptr<D> derived);

void f() {
```

```
  std::shared_ptr<B> poly(new D);
  // ...
  g(std::shared_ptr<D>(dynamic_cast<D *>(poly.get())));
  // Any use of poly will now result in accessing freed memory.
}
```

规范的解决方案

如下代码，通过 dynamic_pointer_cast() 进行转换，它返回一个多态类型智能指针（ std::shared_ptr<B> 类型），该通过赋值构造函数与 poly 关联，避免内存被二次释放。

```
#include <memory>

struct B {
  virtual ~B() = default; // Polymorphic object
  // ...
};
struct D : B {};

void g(std::shared_ptr<D> derived);

void f() {
  std::shared_ptr<B> poly(new D);
  // ...
  g(std::dynamic_pointer_cast<D, B>(poly));
  // poly is still referring to a valid pointer value.
}
```

不合规代码

如下代码中， S::g() 返回的智能指针与存储在 s1 中的智能指针没有关联， s1 与 s2 释放会造成内存双重释放。

```
#include <memory>

struct S {
  std::shared_ptr<S> g() { return std::shared_ptr<S>(this); }
};

void f() {
```

```
  std::shared_ptr<S> s1 = std::make_shared<S>();
  // ...
  std::shared_ptr<S> s2 = s1->g();
}
```

合规代码

如下合规代码中，通过 `shared_from_this()` 保证 `s2` 与 `s1` 通过赋值时相关，从而避免同一内存被双重释放。

```
#include <memory>

struct S : std::enable_shared_from_this<S> {
  std::shared_ptr<S> g() { return shared_from_this(); }
};

void f() {
  std::shared_ptr<S> s1 = std::make_shared<S>();
  std::shared_ptr<S> s2 = s1->g();
}
```

参考文献

- MEM56-CPP. Do not store an already-owned pointer value in an unrelated smart pointer

Rule 07. 输入与输出

FIO50-CPP. 不要在没有定位操作的情况下交替地从同一个文件流中输入输出

作者： 曹笑10116982 评审人： 曾亮亮10091330

basic_filebuf对象控制的读写序列约束与C语言标准库文件操作一致。

C语言标准描述如下:

> *当某文件使用更新模式打开时,相关联的流可以进行同时读写。除非输入操作抵*

> *达文件末尾(EOF)，否则若没有调用fflush函数或其它文件定位函数（fseek, fsetpos, rewind），输入操作后不应该立即紧跟输出操作，输出操作后也不应该立即紧跟输入操作。*
> *因此，以下场景会导致未定义行为：*

- 当文件不处于文件末尾时，先使用输出流，再用输入流。(两者间需调用std::basic_filebuf::seekoff()定位函数)
- 当文件不处于文件末尾时，先使用输入流，再用输出流。(两者间需调用std::basic_filebuf::seekoff()定位函数)
  在std::basic_filebuf类中，没有其它的函数保证了其行为就如调用了对应的C标准库文件定位函数或std::fflush()一样.
  std::basic_ostream::seekp()或者std::basic_istream::seekg()实际上最终调用了std::basic_filebuf::seekoff()来实现文件流定位操作。这两个函数适用于从std::basic_ostream和std::basic_istream继承的std::basic_iostream以及从std::basic_iostream继承的std::fstream，可保证文件缓存处于一个合法的状态用于后续的I/O操作。

不合规代码

该代码在文件末尾写入数据随即从同一文件读入数据。然而，因在格式化输入输出操作间没有对应的文件定位操作，其行为是未定义的。

```
#include <fstream>
#include <string>

void f(const std::string &fileName) {
  std::fstream file(fileName);
  if (!file.is_open()) {
    // Handle error
    return;
  }

  file << "Output some data";
  std::string str;
  file >> str;
}
```

合规代码

该代码通过在输出和输入操作之间调用了std::basic_istream::seekg()函数，消除了未定义行为。

```
#include <fstream>
#include <string>

void f(const std::string &fileName) {
  std::fstream file(fileName);
  if (!file.is_open()) {
    // Handle error
    return;
  }

  file << "Output some data";

  std::string str;
  file.seekg(0, std::ios::beg);
  file >> str;
}
```

参考文献

- FIO50-CPP Do not alternately input and output from a file stream without an intervening positioning call

FIO51-CPP. 不再需要的时候就关闭文件

作者： 曹笑10116982 评审人： 曾亮亮10091330

若调用了std::basic_filebuf::open()，则需要确保在生命期结束或程序正常结束前有对等的std::basic_filebuf::close()调用。
在std::basic_ifstream, std::basic_ofstream, 和std::basic_fstream类中，均持有了一个对std::basic_filebuf的内部引用，来保证在构造函数中调用了open()，析构函数中调用了close()。
一般来说:最佳解决方案是使用值语义的流对象代替动态内存分配，然而当析构函数不能自动调用时仍存在风险。

不合规代码

该代码构造了一个std::fstream对象file，std::fstream的构造函数调用了std::basic_filebuf::open()，std::terminate()函数的默认std::terminate_handler行为是调用std::abort()，但其并未调用析构函数！因此不会调用std::basic_filebuf::close()函数

```
#include <exception>
#include <fstream>
#include <string>

void f(const std::string &fileName) {
  std::fstream file(fileName);
  if (!file.is_open()) {
    // Handle error
    return;
  }
  // ...
  std::terminate();
}
```

合规代码

该代码在std::terminate()调用前调用std::fstream::close()函数，保证了正确关闭文件资源。

```
#include <exception>
#include <fstream>
#include <string>

void f(const std::string &fileName) {
  std::fstream file(fileName);
  if (!file.is_open()) {
    // Handle error
    return;
  }
  // ...
  file.close();
  if (file.fail()) {
    // Handle error
  }
  std::terminate();
}
```

最佳合规代码

该代码中，在std::terminate()调用前通过RAII隐式关闭了文件流，从而保证了正确关闭文件资源。

```
#include <exception>
#include <fstream>
#include <string>

void f(const std::string &fileName) {
  {
    std::fstream file(fileName);
    if (!file.is_open()) {
      // Handle error
      return;
    }
  } // file is closed properly here when it is destroyed
  std::terminate();
}
```

参考文献

- FIO51-CPP Close files when they are no longer needed

Rule 08. 异常与错误处理

ERR50-CPP. 不要突然终止程序

作者： 傅锦华10108953 评审人： 李道春10073354

`std::abort()` ， `std::quick_exit()` 和 `std::_Exit()` 函数以一种立即的方式来终止程序，这些函数不会调用在 `std::atexit()` 注册的退出处理，不会执行对象的自动、线程或静态存储期间的析构方法。
`std::terminate()` 函数会调用当前的 `terminate_handler` 函数，无 `terminate_handler` 函数时默认调用 `std::abort()` .

不合规代码

`f()` 被注册为 `td::at_exit()` 的一个退出处理，调用 `f()` 时，有可能会导致调用 `std::terminate()` ，因为 `throwing_func()` 有可能会抛出一个异常。

```
#include <cstdlib>
```

```
void throwing_func() noexcept(false);

void f() { // Not invoked by the program except as an exit handler.
  throwing_func();
}

int main() {
  if (0 != std::atexit(f)) {
    // Handle error
  }
  // ...
}
```

合规代码

此处 f() 处理了所有由 throwing_func() 抛出的异常，并且没有重新抛出异常。

```
#include <cstdlib>

void throwing_func() noexcept(false);

void f() { // Not invoked by the program except as an exit handler.
  try {
    throwing_func();
  } catch (...) {
    // Handle error
  }
}

int main() {
  if (0 != std::atexit(f)) {
    // Handle error
  }
  // ...
}
```

例外

在遇到一个无法恢复的关键程序错误后，可以显式地调用 std::abort() ，std::_Exit() ，或 std::terminate() 终止程序，比如下面的例子：

```
#include <exception>

void report(const char *msg) noexcept;
[[noreturn]] void fast_fail(const char *msg) {
  // Report error message to operator
  report(msg);

  // Terminate
  std::terminate();
}

void critical_function_that_fails() noexcept(false);

void f() {
  try {
    critical_function_that_fails();
  } catch (...) {
    fast_fail("Critical function failure");
  }
}
```

参考文献

- ERR50-CPP.Do not abruptly terminate the program

ERR51-CPP. 处理所有异常

作者： 傅锦华10108953 评审人： 李道春10073354

当异常抛出时，控制会转移到与抛出异常类型最匹配的处理中去。如果没有直接在异常抛出的try块中找到匹配的处理，搜索匹配处理程序会继续在同一线程的try块中动态搜索匹配的处理。

不合规代码

`f()` 和 `main()` 都没有捕获 `throwing_func()` 抛出的异常。由于找不到对这个抛出异常的匹配的处理， `std::terminate()` 被调用了。

```
void throwing_func() noexcept(false);
```

```
void f() {
  throwing_func();
}

int main() {
  f();
}
```

合规代码

主入口点处理了所有异常，保证了程序被优雅地终止。

```
void throwing_func() noexcept(false);

void f() {
  throwing_func();
}

int main() {
  try {
    f();
  } catch (...) {
    // Handle error
  }
}
```

不合规代码

线程入口函数 `thread_start()` 没有捕获 `throwing_func()` 抛出的异常， `std::terminate()` 会被调用。

```
#include <thread>

void throwing_func() noexcept(false);

void thread_start() {
  throwing_func();
}

void f() {
  std::thread t(thread_start);
```

```
  t.join();
}
```

合规代码

thread_start() 处理了所有异常，并且没有再抛出异常，使得线程能正常终止。

```
#include <thread>

void throwing_func() noexcept(false);

void thread_start(void) {
  try {
    throwing_func();
  } catch (...) {
    // Handle error
  }
}

void f() {
  std::thread t(thread_start);
  t.j
```

参考文献

- ERR51-CPP.Handle all exceptions

ERR52-CPP. 不要使用setjmp()或longjmp()

作者： 傅锦华10108953 评审人： 李道春10073354

C标准库工具 setjmp() 和 longjmp() 可以用来模拟抛出和捕获异常。但是，这些工具会避开自动资源管理，可能导致未定义的行为，通常包括资源泄露和拒绝服务攻击。不要调用 setjmp() 或 longjmp() ，可以用更标准的语句比如 throw 表达式和 catch 语句替换。

不合规代码

如果一个 throw 表达式会导致析构函数被调用，那么在相同的上下文中调用 longjmp()

将导致未定义的行为。在下面的代码例子中， `longjmp()` 的调用出现在一个本地计数器对象的上下文中，会导致未定义的结果。

```
#include <csetjmp>
#include <iostream>

static jmp_buf env;

struct Counter {
  static int instances;
  Counter() { ++instances; }
  ~Counter() { --instances; }
};

int Counter::instances = 0;

void f() {
  Counter c;
  std::cout << "f(): Instances: " << Counter::instances << std::endl;
  std::longjmp(env, 1);
}

int main() {
  std::cout << "Before setjmp(): Instances: " << Counter::instances << std::endl;
  if (setjmp(env) == 0) {
    f();
  } else {
    std::cout << "From longjmp(): Instances: " << Counter::instances << std::endl;
  }
  std::cout << "After longjmp(): Instances: " << Counter::instances << std::endl;
```

执行详细

上面的代码用 `Liunx` 的 `Clang 3.8` 编译会产生如下的结果，证明该程序在这个平台上，当 `f()` 执行终止时，无法销毁本地计数器实例，这个结果是可能的，因为这个行为没有被定义。

```
Before setjmp(): Instances: 0
f(): Instances: 1
From longjmp(): Instances: 1
After longjmp(): Instances: 1
```

合规代码

使用 `throw` 表达式和 `catch` 语句代替了对 `setjmp()` 和 `longjmp()` 的调用。

```
#include <iostream>

struct Counter {
  static int instances;
  Counter() { ++instances; }
  ~Counter() { --instances; }
};

int Counter::instances = 0;

void f() {
  Counter c;
  std::cout << "f(): Instances: " << Counter::instances << std::endl;
  throw "Exception";
}

int main() {
  std::cout << "Before throw: Instances: " << Counter::instances << std::endl;
  try {
    f();
  } catch (const char *E) {
    std::cout << "From catch: Instances: " << Counter::instances << std::endl;
  }
  std::cout << "After catch: Instances: " << Counter::instances << std::endl;
}
```

该代码产生如下的输出：

```
Before throw: Instances: 0
f(): Instances: 1
From catch: Instances: 0
After catch: Instances: 0
```

参考文献

- ERR52-CPP.Do not use setjmp() or longjmp()

ERR53-CPP. 不要在构造或析构方法的function-try-block句柄中引用基类或类的成员变量

作者： 傅锦华10108953 评审人： 李道春10073354

当一个异常在构造方法中被 `function-try-block` 处理捕获时，对象的完全构造基类和类成员会在进入处理程序之前被销毁。类似地，当一个异常在析构方法中被 `function-try-block` 处理捕获时，对象的所有基类和类的成员在进入处理程序之前被摧毁。

不合规代码

类C的构造方法用 `function-try-block` 处理了异常，并在 `catch` 语句中引用了类数据成员，会产生未定义的行为。

```
#include <string>

class C {
  std::string str;

public:
  C(const std::string &s) try : str(s) {
    // ...
  } catch (...) {
    if (!str.empty()) {
      // ...
    }
  }
};
```

合规代码

在 `catch` 语句中引用构造函数参数而不是类数据成员，从而避免未定义的行为。

```
#include <string>

class C {
  std::string str;

public:
  C(const std::string &s) try : str(s) {
```

```
    // ...
  } catch (...) {
    if (!s.empty()) {
      // ...
    }
  }
};
```

参考文献

- ERR53-CPP.Do not reference base classes or class data members in a constructor or destructor function-try-block handler

ERR54-CPP. 异常的捕获句柄应该按参数类型来排序，从最深的派生到最浅的派生

作者： 傅锦华10108953 评审人： 李道春10073354

`Try` 块的处理程序按字面的顺序进行捕获，会导致一些处理程序永远不会被执行，比如在基类的处理程序之后放一个派生类的处理程序。因此，如果两个处理程序捕获来自同一基类的异常（例如 `std::exception` ），则最新的处理捕获异常。

不合规代码

在这个代码例子中，最先的处理捕获了类B的所有异常，然后是类D的所有异常，由于它们都是类B，因此，第二个处理捕获不到任何异常。

```
// Classes used for exception handling
class B {};
class D : public B {};

void f() {
  try {
    // ...
  } catch (B &b) {
    // ...
  } catch (D &d) {
    // ...
  }
}
```

合规代码

在下面的代码中，最先的处理捕获了类D的所有异常，接下来的处理捕获了类B的所有其他异常。

```
// Classes used for exception handling
class B {};
class D : public B {};

void f() {
  try {
    // ...
  } catch (D &d) {
    // ...
  } catch (B &b) {
    // ...
  }
}
```

参考文献

- ERR54-CPP.Catch handlers should order their parameter types from most derived to least derived

ERR55-CPP. 规范异常说明

作者： 傅锦华10108953 评审人： 李道春10073354

如果一个函数抛出一个异常规范允许外的异常，会导致程序的应用定义终止。
如果一个声明了动态异常规范的函数抛出一个与异常规范不匹配的异常，那么函数 `std::unexpected()` 会被调用。类似地，如果一个声明了无异常规范的函数抛出了一个导致无异常规范求值为 `False` 的异常，那么函数 `std::terminate()` 会被调用，导致程序的应用定义终止。
为了防止程序的异常终止，任何声明了一个异常规范的函数必须严格约束自己，在调用任何函数时，只抛出允许的异常。

不合规代码

下面的代码中，第二个函数只说明了抛出 `Exception1` ，但是他有可能会抛出 `Exception2` .

```
#include <exception>

class Exception1 : public std::exception {};
class Exception2 : public std::exception {};

void foo() {
  throw Exception2{}; // Okay because foo() promises nothing about exceptions
}

void bar() throw (Exception1) {
  foo();    // Bad because foo() can throw Exception2
}
```

合规代码

下面的代码捕获了 `foo()` 函数抛出的异常。

```
#include <exception>

class Exception1 : public std::exception {};
class Exception2 : public std::exception {};

void foo() {
  throw Exception2{}; // Okay because foo() promises nothing about exceptions
}

void bar() throw (Exception1) {
  try {
    foo();
  } catch (Exception2 e) {
    // Handle error without rethrowing it
  }
}
```

合规代码

下面的代码声明了 `bar()` 函数的动态异常规范，覆盖了所有可能抛出的异常。

```
#include <exception>

class Exception1 : public std::exception {};
class Exception2 : public std::exception {};

void foo() {
  throw Exception2{}; // Okay because foo() promises nothing about exceptions
}

void bar() throw (Exception1, Exception2) {
  foo();
}
```

不合规代码

下面的代码中，函数被声明没有异常抛出，但是有可能在请求的内存无法分配时导致 `std::vector::resize()` 抛出一个异常。

```
#include <cstddef>
#include <vector>

void f(std::vector<int> &v, size_t s) noexcept(true) {
  v.resize(s); // May throw
}
```

合规代码

下面的代码中，函数的无异常规范被移除，表明该函数允许任何异常。

```
#include <cstddef>
#include <vector>

void f(std::vector<int> &v, size_t s) {
  v.resize(s); // May throw, but that is okay
}
```

执行详细

一些厂商提供语言扩展来指定函数是否抛出。比如，微软的 `Visual Studio` 提供了 `__declspec(nothrow))` ， `Clang` 支持 `__attribute__((nothrow))` 。目前，这些厂商

没有使用这些扩展来注解一个函数不抛出异常的行为。从使用这些运用扩展的函数抛出的的异常被假定为未定义行为。

参考文献

- ERR55-CPP.Honor exception specifications

ERR56-CPP. 保证异常能被安全地处理

作者： 傅锦华10108953 评审人： 李道春10073354

正确处理错误和异常情况对软件的持续正确运行至关重要。C++程序中报告错误的优先机制是异常而不是错误代码。
下表区分了三种异常安全保证，从最需要的到最不需要的。

保证	描述	例子
强	强异常安全保证了这样的操作属性，除了满足基本异常安全保证外，如果该操作通过抛出异常终止，则对程序状态没有可观察到的影响。	强异常安全
基本	基本异常安全保证了这样的操作属性，如果操作通过抛出异常来终止，则需要保留程序状态不变，防止资源泄露。	基本异常安全
空	非异常安全指既不提供强也非基本异常安全保证的代码。	非异常安全

保证强异常安全的代码也保证基本异常安全。
不提供异常安全保证的代码是不安全的，是有缺陷的。

不合规代码(无异常安全)

下面的代码示例展示了一个有缺陷的不符合要求的拷贝赋值运算符。 `array` 和 `nelems` 是类的隐藏变量，其中 `array` 成员是一个有效的（可能为空）的指针， `nelems` 成员存储了指向这个数组的数组元素个数。在为副本分配新内存块之前，函数释放了 `array` 并且给 `nelems` 赋值了元素个数。因此，如果 `new` 表达式抛出异常，该函数将修改两个成员变量的状态，这就违反了类的隐式不变性。也就是说，对不确定状态下的对象的任何操

作，包括它的析构，都会引发未定义行为。

```
#include <cstring>

class IntArray {
  int *array;
  std::size_t nElems;
public:
  // ...

  ~IntArray() {
    delete[] array;
  }


  IntArray(const IntArray& that); // nontrivial copy constructor
  IntArray& operator=(const IntArray &rhs) {
    if (this != &rhs) {
      delete[] array;
      array = nullptr;
      nElems = rhs.nElems;
      if (nElems) {
        array = new int[nElems];
        std::memcpy(array, rhs.array, nElems * sizeof(*array));
      }
    }
    return *this;
  }

  // ...
```

合规代码(强异常安全)

下面的代码中，拷贝赋值操作符提供了强异常安全保证。该函数在更改对象的状态之前为副本分配新的存储区。只有在分配成功后，函数才会继续改变对象的状态。此外，把拷贝数组到新的存储区之前释放已存在的数组，函数避免了自分配测试，这通常会提升代码的性能。

```
#include <cstring>

class IntArray {
  int *array;
```

```
  std::size_t nElems;
public:
  // ...

  ~IntArray() {
    delete[] array;
  }

  IntArray(const IntArray& that); // nontrivial copy constructor

  IntArray& operator=(const IntArray &rhs) {
    int *tmp = nullptr;
    if (rhs.nElems) {
      tmp = new int[rhs.nElems];
      std::memcpy(tmp, rhs.array, rhs.nElems * sizeof(*array));
    }
    delete[] array;
    array = tmp;
    nElems = rhs.nElems;
    return *this;
  }

  // ...
};
```

参考文献

- ERR56-CPP.Guarantee exception safety

ERR57-CPP. 不要在处理异常时泄露资源

作者： 傅锦华10108953 评审人： 李道春10073354

在异常抛出时回收资源是很重要的。抛出的异常可能会导致清除那些正在被传递或对象处于部分初始化状态的代码。

不合规代码

下面代码中，当 `process_item` 抛出一个异常， `pst` 没有被正确地释放，从而导致了资源泄露。

```
#include <new>

struct SomeType {
  SomeType() noexcept; // Performs nontrivial initialization.
  ~SomeType(); // Performs nontrivial finalization.
  void process_item() noexcept(false);
};

void f() {
  SomeType *pst = new (std::nothrow) SomeType();
  if (!pst) {
    // Handle error
    return;
  }

  try {
    pst->process_item();
  } catch (...) {
    // Process error, but do not recover from it; rethrow.
    throw;
  }
  delete pst;
}
```

合规代码(删除)

下面的代码中，异常处理通过调用 `delete` 释放了 `pst` 。

```
#include <new>

struct SomeType {
  SomeType() noexcept; // Performs nontrivial initialization.
  ~SomeType(); // Performs nontrivial finalization.

  void process_item() noexcept(false);
};

void f() {
  SomeType *pst = new (std::nothrow) SomeType();
  if (!pst) {
    // Handle error
    return;
```

```
  }
  try {
    pst->process_item();
  } catch (...) {
    // Process error, but do not recover from it; rethrow.
    delete pst;
    throw;
  }
  delete pst;
}
```

虽然这个代码使用 `catch` 语句正确地释放它的资源，这种方法可能有一些缺点：

- 每一个不同的清理需要自己的 `try` 和 `catch` 块。
- 清理操作不能抛出任何异常。

合规代码(RAII设计模式)

一个更好的方法是采用RAII, 这种模式迫使每一个对象在遇到异常行为后都会自我清理，防止程序员不得不这样做，这样做的另一个好处是它不需要代码来处理资源分配错误。

```
struct SomeType {
  SomeType() noexcept; // Performs nontrivial initialization.
  ~SomeType(); // Performs nontrivial finalization.

  void process_item() noexcept(false);
};

void f() {
  SomeType st;
  try {
    st.process_item();
  } catch (...) {
    // Process error, but do not recover from it; rethrow.
    throw;
  } // After re-throwing the exception, the destructor is run for st.
} // If f() exits without throwing an exception, the destructor is run for st.
```

不合规代码

下面的代码中，构造方法 `C::C()` 可能会在给 `a` 分配内存时出错，也有可能在给 `b` 分配内存时出错，也可能在 `init()` 方法中抛出一个异常。如果 `init()` 抛出一个异常，那

么 a 和 b 都不会被释放。同样，如果 b 的分配失败， a 将不会被释放。

```
struct A {/* ... */};
struct B {/* ... */};

class C {
  A *a;
  B *b;
protected:
  void init() noexcept(false);
public:
  C() : a(new A()), b(new B()) {
    init();
  }
};
```

合规代码(try/catch)

这段代码通过在构造或 init() 过程中抛出异常时释放 a 和 b ，减少了潜在的失败。

```
struct A {/* ... */};
struct B {/* ... */};

class C {
  A *a;
  B *b;
protected:
  void init() noexcept(false);
public:
  C() : a(nullptr), b(nullptr) {
    try {
      a = new A();
      b = new B();
      init();
    } catch (...) {
      delete a;
      delete b;
      throw;
    }
  }
};
```

合规代码(std::unique_ptr)

下面的代码使用 `std::unique_ptr` 来创建对象，该对象在 `C::C()` 中有任何错误时进行清理操作。 `std::unique_ptr` 应用RAII指针原则。

```
#include <memory>

struct A {/* ... */};
struct B {/* ... */};

class C {
  std::unique_ptr<A> a;
  std::unique_ptr<B> b;
protected:
  void init() noexcept(false);
public:
  C() : a(new A()), b(new B()) {
    init();
  }
};
```

参考文献

- ERR57-CPP.Do not leak resources when handling exceptions

ERR58-CPP. 在main方法开始处理时处理所有的异常抛出

作者： 傅锦华10108953 评审人： 李道春10073354

即使很小心地使用function-try-blocks，也不能捕获所有的异常。

- *静态存储期间对象的析构方法或命名空间范围静态存储期间对象的析构方法中抛出的异常不会被main()里面的function-try-block 捕获。*
- *静态存储期间对象的析构方法或命名空间范围静态存储期间对象的析构方法中抛出的异常不会被线程初始化方法里面的function-try-block 捕获。*

不合规代码

下面的代码中， `s` 的构造函数可能会抛出一个异常，在程序启动过程中， `globalS` 被构

造时，该异常不会被捕获。

```
struct S {
  S() noexcept(false);
};

static S globalS;
```

合规代码

下面的代码中，把 globalS 放到具有静态存储期间的局部变量中，这样由于 s 的构造函数会在第一次调用 globalS() 方法时被调用，而不是在程序启动时被调用，这样就允许在对象的构造方法中捕获任何异常。这个解决方案不需要程序员修改代码，只要把以前对 globalS 变量的使用替换成对 globalS() 函数的调用。

```
struct S {
  S() noexcept(false);
};

S &globalS() {
  try {
    static S s;
    return s;
  } catch (...) {
    // Handle error, perhaps by logging it and gracefully terminating the applicatio
n.
  }
  // Unreachable.
}
```

不合规代码

下面的代码中， global 的构造函数可能会在程序启动过程中抛出一个异常。（ std::string 的构造函数，接受一个 const char * 的变量作为默认构造对象，没有被标记为无异常，因此允许任何异常）。该异常没有被 main() 方法的 function-try-block 捕获，导致 std::terminate() 被调用，程序异常终止。

```
#include <string>

static const std::string global("...");
```

```
int main()
try {
  // ...
} catch(...) {
  // IMPORTANT: Will not catch exceptions thrown
  // from the constructor of global
}
```

合规代码

合规代码必须防止异常在程序启动或终止过程中被抛出。下面的代码避免在全局命名空间范围内定义一个 `std::string` 变量，取而代之使用一个 `static const char *` 变量。

```
static const char *global = "...";

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

合规代码

下面的代码引入了一个从 `std::string` 衍生的类，该类构造时通过函数的 `try` 块捕获了所有异常，并在任何异常抛出的 `catch` 块中终止程序。（简单起见，没有描述对这样一个类型的全部接口）

```
#include <exception>
#include <string>

namespace my {
struct string : std::string {
  explicit string(const char *msg,
                  const std::string::allocator_type &alloc = std::string::allocator_
type{}) noexcept
  try : std::string(msg, alloc) {} catch(...) {
    extern void log_message(const char *) noexcept;
    log_message("std::string constructor threw an exception");
    std::terminate();
  }
  // ...
};
```

```
}

static const my::string global("...");

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

不合规代码

下面的代码中，异常有可能在对静态全局变量 `i` 初始化的方法中被抛出。

```
extern int f() noexcept(false);
int i = f();

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

合规代码

下面的代码中，把对 `f()` 的调用封装到了一个 `helper` 函数中，该函数捕获了所有的异常，并在 `catch` 块中终止了程序。

```
#include <exception>

int f_helper() noexcept {
  try {
    extern int f() noexcept(false);
    return f();
  } catch (...) {
    extern void log_message(const char *) noexcept;
    log_message("f() threw an exception");
    std::terminate();
  }
  // Unreachable.
}

int i = f_helper();

int main() {
```

```
  // ...
}
```

参考文献

- ERR58-CPP.Handle all exceptions thrown before main() begins executing

ERR59-CPP. 不要在执行边界上抛出异常

作者： 傅锦华10108953 评审人： 李道春10073354

抛出异常需要执行代码与catch语句的协作。然而，当在执行边界上抛出异常时，必须注意确保运行时逻辑在执行边界的不同边之间是兼容的。
一个执行边界是由不同编译器编译的代码之间的界限，包括由同一个厂商生产的不同版本的编译器。例如，一个函数可能在头文件中声明，但定义在运行时加载的库中。执行边界位于可执行文件中的调用点与库中的函数实现之间。这样的界限也被称为ABI（应用二进制接口）的界限，因为它们涉及到应用程序二进制文件的互操作性。

只有当执行边界双方使用相同的ABI时才能抛出一个执行边界异常。

不合规代码

下面的代码中，从一个库函数抛出一个异常。尽管这个异常是一个标量类型，但是如果库和应用遵循不同的ABIs，这段代码仍然可能导致程序执行异常。

```
// library.h
void func() noexcept(false); // Implemented by the library

// library.cpp
void func() noexcept(false) {
  // ...
  if (/* ... */) {
    throw 42;
  }
}

// application.cpp
#include "library.h"

void f() {
```

```
  try {
    func();
  } catch(int &e) {
    // Handle error
  }
}
```

如果使用的库代码是在一个默认安装 MinGW-w64 的环境汇总，使用 GCC 4.9 编译的，没有特殊编译标志，那么异常的抛出讲依赖基于 DWARF 和 Itanium ABI 的 zero-cost，table-based 异常模型。如果应用程序代码使用 Visual Studio 2013 编译的， catch 处理程序将基于结构化异常处理和微软 ABI 。这两个异常处理机制是不兼容的，虽然都是 ABIs ，仍会导致程序执行异常。具体而言，由库引发的异常不会被应用捕获，但 std::terminate() 会被调用。

合规代码

下面的代码中，库函数的错误由返回值来表示，而不是异常。库和应用都使用微软 Visual Studio （或 GCC ）编译也是一个兼容的解决方案，因为在两边的执行边界使用相同的异常处理机制和 ABI 。

```
// library.h
int func() noexcept(true); // Implemented by the library

// library.cpp
int func() noexcept(true) {
  // ...
  if (/* ... */) {
    return 42;
  }
  // ...
  return 0;
}

// application.cpp
#include "library.h"

void f() {
  if (int err = func()) {
    // Handle error
  }
}
```

参考文献

- ERR59-CPP.Do not throw an exception across execution boundaries

ERR60-CPP. 异常对象必须是无异常抛出的拷贝构造

作者： 傅锦华10108953 评审人： 李道春10073354

当抛出异常时，抛出表达式的异常对象操作数将被复制到用于初始化处理程序的临时对象。
如果在执行复制构造函数时抛出异常，会调用 `std::terminate()` ，此时的执行结果是未定义的。

不合规代码

下面的代码中，一个 `S` 类型的异常从 `f()` 中抛出。然而，由于 `S` 有一个 `std::string` 类型的数据成员，并且 `std::string` 类型的拷贝构造函数未定义成非异常。在低内存的情况下，对 `std::string` 的拷贝构造函数可能无法分配足够的内存来完成拷贝操作，从而导致 `std::bad_alloc` 异常被抛出。

```
#include <exception>
#include <string>

class S : public std::exception {
  std::string m;
public:
  S(const char *msg) : m(msg) {}

  const char *what() const noexcept override {
    return m.c_str();
  }
};

void g() {
  // If some condition doesn't hold...
  throw S("Condition did not hold");
}

void f() {
  try {
```

```
    g();
  } catch (S &s) {
    // Handle error
  }
}
```

合规代码

下面的代码假设异常对象的类型可以从 std::runtime_error 继承，或者这个类型可以直接使用。不像 std::string ， std::runtime_error 对象要求正确处理任意长度错误消息，这样是安全的，并且保证拷贝构造函数不会抛错。

```
#include <stdexcept>
#include <type_traits>

struct S : std::runtime_error {
  S(const char *msg) : std::runtime_error(msg) {}
};

static_assert(std::is_nothrow_copy_constructible<S>::value,
              "S must be nothrow copy constructible");

void g() {
  // If some condition doesn't hold...
  throw S("Condition did not hold");
}

void f() {
  try {
    g();
  } catch (S &s) {
    // Handle error
  }
}
```

合规代码

如果异常类型不能修改为从 std::runtime_error 继承，可以加入一个 std::runtime_error 类型的数据成员，这也是一个合法的策略，如下：

```
#include <stdexcept>
```

```
#include <type_traits>

class S : public std::exception {
  std::runtime_error m;
public:
  S(const char *msg) : m(msg) {}

  const char *what() const noexcept override {
    return m.what();
  }
};

static_assert(std::is_nothrow_copy_constructible<S>::value,
              "S must be nothrow copy constructible");

void g() {
  // If some condition doesn't hold...
  throw S("Condition did not hold");
}

void f() {
  try {
    g();
  } catch (S &s) {
    // Handle error
  }
}
```

参考文献

- ERR60-CPP.Exception objects must be nothrow copy constructible

ERR61-CPP. 通过左值引用来捕获异常

作者： 傅锦华10108953 评审人： 李道春10073354

除非是平凡的类型，应该总是通过左值引用捕获异常。
C++标准定义的平凡的类型如下：
算术类型，枚举类型，指针类型，指向成员的指针类型， `std::nullptr_t` ，和这些类型的版本统称为标量类型的...标量类型、基本的类类型、此类类型的数组统称为平凡类型。
C++标准定义的平凡的类类型如下：

一个平凡的类是指一个类有一个默认构造函数，没有非平凡的默认构造函数，并且是一个平凡的可拷贝类。
一个平凡的可拷贝类是指：

- 没有非平凡的拷贝构造函数
- 没有非平凡的移动构造函数
- 没有非平凡的拷贝赋值操作符
- 没有非平凡的移动赋值运算符
- 有一个平凡的析构函数

不合规代码

下面的代码中，类型 `S` 的对象用来初始化 `std::exception` 声明的异常对象。异常声明匹配异常对象类型，所以变量 `E` 是从异常对象拷贝初始化，导致异常对象被切片。因此，这个代码示例的输出是 `std::exception::what()` ，而不是 “My custom exception” .

```
#include <exception>
#include <iostream>

struct S : std::exception {
  const char *what() const noexcept override {
    return "My custom exception";
  }
};

void f() {
  try {
    throw S();
  } catch (std::exception e) {
    std::cout << e.what() << std::endl;
  }
}
```

合规代码

下面的代码中，异常变量的声明是左值引用。调用 `what()` 结果是执行 `S::what()` ，而不是 `std::exception::what()` .

```
#include <exception>
#include <iostream>
```

```
struct S : std::exception {
  const char *what() const noexcept override {
    return "My custom exception";
  }
};

void f() {
  try {
    throw S();
  } catch (std::exception &e) {
    std::cout << e.what() << std::endl;
  }
}
```

参考文献

- ERR61-CPP.Exception objects must be nothrow copy constructible

Rule 09. 面向对象编程

OOP50-CPP. 禁止在构造函数或析构函数中调用虚函数

作者： 刘光聪10209986 评审人： 李永顺10110636

当在对象构造或者析构期间，不要在构造函数或者析构函数中直接或间接调用虚函数。因为对象是按基类到子类的顺序构造的，在构造期间尝试从基类中调用子类的函数是相当危险的。此时，子类还没有机会去完成资源的初始化；如果在构造函数中调用虚函数，那么调用将不会发生在子类上。
同样地，对象析构的顺序与构造的顺序相反。当在子类的析构函数中调用虚函数，则有可能非法获取已经释放了的资源。

不合规代码

在此处的不合规代码中，基类尝试在构造函数和析构函数中分别调用 `seize` 与 `release` 的虚函数。但是， `B::B()` 构造函数实际上调用了 `B::seize()` ，而不是 `D::seize()` 。同样地， `B::~B()` 析构函数调用了 `B::release()` ，而不是 `D::release()` .

```
struct B {
  B() { seize(); }
  virtual ~B() { release(); }

protected:
  virtual void seize();
  virtual void release();
};

struct D : B {
  virtual ~D() = default;

protected:
  void seize() override {
    B::seize();
    // Get derived resources...
  }

  void release() override {
    // Release derived resources...
    B::release();
  }
};
```

合规代码

在此处的合规代码中，构造函数和析构函数调用了非虚的、私有的成员函数(以 `mine` 为后缀)，替代原来的虚函数调用，结果每个类各自承担所持有的资源的分配和释放的职责。

```
class B {
  void seize_mine();
  void release_mine();

public:
  B() { seize_mine(); }
  virtual ~B() { release_mine(); }

protected:
  virtual void seize() { seize_mine(); }
  virtual void release() { release_mine(); }
};
```

```
class D : public B {
  void seize_mine();
  void release_mine();

public:
  D() { seize_mine(); }
  virtual ~D() { release_mine(); }

protected:
  void seize() override {
    B::seize();
    seize_mine();
  }

  void release() override {
    release_mine();
    B::release();
  }
};
```

例外

存在一些合理的场景，允许在构造函数中通过显式地指定类的方式来调用虚函数(不是纯虚的)。

- 情况1: 通过显式的类标识符明确地告诉代码维护者，此处期待完成在构造函数或析构函数中函数的静态调用。

```
struct A {
  A() {
    // f();   // WRONG!
    A::f();   // Okay
  }
  virtual void f();
};
```

- 情况2: 可以在构造函数或析构函数中调用一个 `final` 修饰的虚函数。

```
struct A {
  A();
  virtual void f();
};
```

```
struct B : A {
  B() : A() {
    f();  // Okay
  }
  void f() override final;
};
```

- 情况3: 同样地，当一个类被修饰为 `final` 时，可以在其构造函数或析构函数中调用虚函数。

```
struct A {
  A();
  virtual void f();
};

struct B final : A {
  B() : A() {
    f();  // Okay
  }
  void f() override;
};
```

在其他场景， `f()` 必须使用 `final override` 修饰，保证函数调用行为的一致性。

参考文献

- Do not invoke virtual functions from constructors or destructors

OOP51-CPP. 禁止切割子类对象

作者： 刘光聪10209986 评审人： 李永顺10110636

当子类继承父类，子类对象一般会附加其他成员变量。当以值的方式赋值或拷贝子类对象到父类对象时，这些附加的成员变量将不会被拷贝，这个行为通常称为对象切割。
禁止使用子类对象初始化一个父类对象，除非通过引用，指针，或者智能指针的抽象(例如 `std::unique_ptr` ，或者 `std::shared_ptr`)。

不合规代码

在此不合规代码中，子类 Manager 对象通过值的方式传递给一个接受父类 Employee 的函数中。最终， Manager 对象被切割，当调用 print() 函数时，导致信息丢失。

```
#include <iostream>
#include <string>

class Employee {
  std::string name;

protected:
  virtual void print(std::ostream &os) const {
    os << "Employee: " << get_name() << std::endl;
  }

public:
  Employee(const std::string &name) : name(name) {}
  const std::string &get_name() const { return name; }
  friend std::ostream &operator<<(std::ostream &os, const Employee &e) {
    e.print(os);
    return os;
  }
};

class Manager : public Employee {
  Employee assistant;

protected:
  void print(std::ostream &os) const override {
    os << "Manager: " << get_name() << std::endl;
    os << "Assistant: " << std::endl << "\t"
       << get_assistant() << std::endl;
  }

public:
  Manager( const std::string &name
         , const Employee &assistant)
         : Employee(name)
         , assistant(assistant)
  {}

  const Employee &get_assistant() const { return assistant; }
};
```

```
void f(Employee e) {
  std::cout << e;
}

int main() {
  Employee coder("Joe Smith");
  Employee typist("Bill Jones");
  Manager designer("Jane Doe", typist);

  f(coder);
  f(typist);
  f(designer);
}
```

当f()传递参数 `designer` 时， `f()` 的参数将被切割，并导致信息丢失。当打印对象 `e` 时， `Employee::print()` 被调用，而不是 `Manager::print()` ，其运行结果如下：

```
Employee: Jane Doe
```

合规代码(指针)

存在一个改进的方案，使用引用替代指针，在 `f()` 实现里可以消除空指针的保护。

```
// ... Remainder of code unchanged ...

void f(const Employee &e) {
  std::cout << e;
}

int main() {
  Employee coder("Joe Smith");
  Employee typist("Bill Jones");
  Manager designer("Jane Doe", typist);

  f(coder);
  f(typist);
  f(designer);
}
```

合规代码(Noncopyable)

在本合规代码中，通过继承 Noncopyable 抑制对象拷贝构造的能力，避免不经意地发生对象切割的问题。但是，这个解决方案也限制了 Manager 对象拷贝构造 Employee 对象的能力，其微妙地改变了类层次的语义关系。

```
#include <iostream>
#include <string>

class Noncopyable {
  Noncopyable(const Noncopyable &) = delete;
  void operator=(const Noncopyable &) = delete;

protected:
  Noncopyable() = default;
};

class Employee : Noncopyable {
  // Remainder of the definition is unchanged.
  std::string name;

protected:
  virtual void print(std::ostream &os) const {
    os << "Employee: " << get_name() << std::endl;
  }

public:
  Employee(const std::string &name) : name(name) {}
  const std::string &get_name() const { return name; }
  friend std::ostream &operator<<(std::ostream &os, const Employee &e) {
    e.print(os);
    return os;
  }
};

class Manager : public Employee {
  const Employee &assistant; // Note: The definition of Employee has been modified.

  // Remainder of the definition is unchanged.
protected:
  void print(std::ostream &os) const override {
    os << "Manager: " << get_name() << std::endl;
    os << "Assistant: " << std::endl << "\t" << get_assistant() << std::endl;
  }
```

```
public:
  Manager(const std::string &name, const Employee &assistant) : Employee(name), assi
stant(assistant) {}
  const Employee &get_assistant() const { return assistant; }
};

// If f() were declared as accepting an Employee, the program would be
// ill-formed because Employee cannot be copy-initialized.
void f(const Employee &e) {
  std::cout << e;
}

int main() {
  Employee coder("Joe Smith");
  Employee typist("Bill Jones");
  Manager designer("Jane Doe", typist);

  f(coder);
  f(typist);
  f(designer);
}
```

不合规代码

在此不合规代码中，使用了上述合规代码中 `Employee` 和 `Manager` 的类定义，并尝试缓存 `Employee` 对象到 `std::vector` 容器中去。但是， `std::vector` 要求列表元素类型是同质的，其结果将导致对象切割的问题。

```
#include <iostream>
#include <string>
#include <vector>

void f(const std::vector<Employee> &v) {
  for (const auto &e : v) {
    std::cout << e;
  }
}

int main() {
  Employee typist("Joe Smith");
  std::vector<Employee> v{typist, Employee("Bill Jones"), Manager("Jane Doe", typist
)};
```

```
  f(v);
}
```

合规代码

在此合规代码中，使用了持有 `std::unique_ptr` 元素类型的 `vector` 类型，避免了对象切割问题。

```
#include <iostream>
#include <memory>
#include <string>
#include <vector>

void f(const std::vector<std::unique_ptr<Employee>> &v) {
  for (const auto &e : v) {
    std::cout << *e;
  }
}

int main() {
  std::vector<std::unique_ptr<Employee>> v;

  v.emplace_back(new Employee("Joe Smith"));
  v.emplace_back(new Employee("Bill Jones"));
  v.emplace_back(new Manager("Jane Doe", *v.front()));

  f(v);
}
```

参考文献

- Do not slice derived objects

OOP52-CPP. 禁止删除没有虚拟析构函数的多态对象

作者： 刘光聪10209986 评审人： 李永顺10110636

当父类没有定义虚拟析构函数时，尝试使用指向父类的指针删除子类对象将导致未定义的行为。

不合规代码

在此不合规代码中， `b` 是一个多态指针，其静态类型为 `Base*` ，而其动态类型为 `Derived*` 。因为 `Base` 未定义虚拟的析构函数，因此当 `b` 被删除时，其行为是未定义。

```
struct Base {
  virtual void f();
};

struct Derived : Base {};

void f() {
  Base *b = new Derived();
  // ...
  delete b;
}
```

不合规代码

在此不合规代码中，使用智能指针代替原生的指针类型。因为 `std::unique_ptr` 默认的析构函数将会 `delete` 内部持有的原生指针对象，其程序行为与上例类同。

```
#include <memory>
struct Base {
  virtual void f();
};

struct Derived : Base {};

void f() {
  std::unique_ptr<Base> b = std::make_unique<Derived()>();
}
```

合规代码

在此合规代码中，父类显式地声明了虚拟析构函数，保证了 `delete` 多态操作的行为是良好定义的。

```
struct Base {
```

```
  virtual ~Base() = default;
  virtual void f();
};

struct Derived : Base {};

void f() {
  Base *b = new Derived();
  // ...
  delete b;
}
```

参考文献

- Do not delete a polymorphic object without a virtual destructor

OOP53-CPP. 规范化成员初始化列表的顺序

作者： 刘光聪10209986 评审人： 李永顺10110636

成员初始化的顺序与成员初始化列表指定的顺序无关。如果成员初始化未规范化，将导致未定义的行为，例如读取未初始化的内存。
因此，务必需要规范化成员初始化列表的顺序：首先，直接父类按照父类定义列表的顺序进行初始化；然后，非静态成员变量按照类定义的顺序进行初始化。

不合规代码

在此不合规代码中， `C::C()` 的初始化列表先尝试初始化 `someVal` ，然后初始化 `dependsOnSomeVal` 。其中 `dependsOnSomeVal` 的初始化依赖于 `someVal` 。因为成员变量声明的顺序与初始化列表成员的顺序不匹配，尝试读取 `someVal` 的值，并对 `dependsOnSomeVal` 进行初始化，将得到不确定的值。

```
class C {
  int dependsOnSomeVal;
  int someVal;

public:
  C(int val) : someVal(val), dependsOnSomeVal(someVal + 1) {}
};
```

合规代码

在本合规代码中，改变了成员变量的声明顺序，以便保证其顺序与构造函数的初始化列表的顺序保持一致。

```
class C {
  int someVal;
  int dependsOnSomeVal;

public:
  C(int val) : someVal(val), dependsOnSomeVal(someVal + 1) {}
};
```

不合规代码

在该不合规代码中，子类 `D` 尝试初始化基类 `B1` 时，其初始化值取自另外一个基类 `B2` 。但是，根据子类 `D` 的基类定义列表， `B1` 在 `B2` 之前初始化，这将导致程序行为是未定义的。

```
class B1 {
  int val;

public:
  B1(int val) : val(val) {}
};

class B2 {
  int otherVal;

public:
  B2(int otherVal) : otherVal(otherVal) {}
  int get_other_val() const { return otherVal; }
};

class D : B1, B2 {
public:
  D(int a) : B2(a), B1(get_other_val()) {}
};
```

合规代码

在此合规代码中，两个基类使用来自子类构造函数的参数，并取得相同的值。其初始化列表不依赖与基类的初始化顺序。

```
class B1 {
  int val;

public:
  B1(int val) : val(val) {}
};

class B2 {
  int otherVal;

public:
  B2(int otherVal) : otherVal(otherVal) {}
};

class D : B1, B2 {
public:
  D(int a) : B1(a), B2(a) {}
};
```

参考文献

- Write constructor member initializers in the canonical order

OOP54-CPP. 优雅地处理“自我赋值”

作者： 刘光聪10209986 评审人： 李永顺10110636

用户自定义拷贝赋值重载运算符时，必须阻止自我赋值使得对象处于不确定的状态。可以通过自我赋值的测试，拷贝-交换，或其他一些设计模式实现。

不合规代码

在该不合规代码中，拷贝赋值重载运算符并没有实施自我赋值的保护。如果自我赋值发生了， `this->s1` 被销毁， `rhs.s1` 也会被销毁。 `rhs.s1` 不合理的内存被传递给了 `S` 的构造函数，这将会引发解引用一个非法指针的错误。

```
#include <new>
```

```
struct S { /* ... */ }; // Has nonthrowing copy constructor

class T {
  int n;
  S *s1;

public:
  T(const T &rhs) : n(rhs.n), s1(rhs.s1 ? new S(*rhs.s1) : nullptr) {}
  ~T() { delete s1; }

  // ...

  T& operator=(const T &rhs) {
    n = rhs.n;
    delete s1;
    s1 = new S(*rhs.s1);
    return *this;
  }
};
```

合规代码(自我测试)

在此合规代码中，通过测试参数与 `this` 是否是同一对象，以此保护自我赋值的问题。如果自我赋值问题发生了， `operator=` 不会做任何事情。

```
#include <new>

struct S { /* ... */ }; // Has nonthrowing copy constructor

class T {
  int n;
  S *s1;

public:
  T(const T &rhs) : n(rhs.n), s1(rhs.s1 ? new S(*rhs.s1) : nullptr) {}
  ~T() { delete s1; }

  // ...

  T& operator=(const T &rhs) {
    if (this != &rhs) {
```

```
      n = rhs.n;
      delete s1;
      try {
        s1 = new S(*rhs.s1);
      } catch (std::bad_alloc &) {
        s1 = nullptr; // For basic exception guarantees
        throw;
      }
    }
    return *this;
  }
};
```

这个方案对于拷贝赋值并没有提供鲁棒的异常保护机制。确切地说，当执行 `new` 表达式时抛出了异常， `this` 所指对象已经被修改。但是，这个方案提供了基本的异常保护机制，因为没有泄露任何资源，并且所有成员持有合理的值。

合规代码(拷贝-交换)

在此合规代码中，通过 `rhs` 构建一个临时对象，然后再与 `*this` 进行交换，从而避免了自我赋值。当构造临时对象时，资源分配所抛出的异常会导致 `swap()` 不会被调用，因此该方案提供了鲁棒的异常保护机制。

```
#include <new>
#include <utility>

struct S { /* ... */ }; // Has nonthrowing copy constructor

class T {
  int n;
  S *s1;

public:
  T(const T &rhs) : n(rhs.n), s1(rhs.s1 ? new S(*rhs.s1) : nullptr) {}
  ~T() { delete s1; }

  // ...

  void swap(T &rhs) noexcept {
    using std::swap;
    swap(n, rhs.n);
    swap(s1, rhs.s1);
```

```
  }

  T& operator=(const T &rhs) noexcept {
    T(rhs).swap(*this);
    return *this;
  }
};
```

参考文献

- Gracefully handle self-copy assignment

OOP55-CPP. 禁止使用指向成员的指针访问不存在的成员

作者： 刘光聪10209986 评审人： 李永顺10110636

指向成员的操作符 `.*` 与 `->*` ，用于获取一个对象的成员变量或成员函数。例如，在对象 `o` 上调用成员函数 `f()` ，如下方法功能等价。

```
struct S {
  void f() {}
};

void func() {
  S o;
  void (S::*pm)() = &S::f;

  o.f();
  (o.*pm)();
}
```

`o.f()` 的调用方式，在编译期通过获取对象 `o` 的成员函数 `S::f()` 的地址，直接获取类成员。但是，由 `pm` 指定的 `(o.*pm)` 函数调用方式，使用指向成员的指针操作符 `.*` ，通过 `pm` 所指定的地址间接调用函数。

不合规代码

在此不合规代码中，通过 `D::g` 获取了一个指向成员的指针，但是被向上强制转型

为 `B::*` 。当在一个动态类型为 `D` 的对象上调用该指向成员的指针时，其行为是被良好定义的。但是，如果在一个动态类型为 `B` 的对象上调用该指向成员的指针时，其行为是未定义的。

```
struct B {
  virtual ~B() = default;
};

struct D : B {
  virtual ~D() = default;
  virtual void g() { /* ... */ }
};

void f() {
  B *b = new B;

  // ...

  void (B::*gptr)() = static_cast<void(B::*)()>(&D::g);
  (b->*gptr)();
  delete b;
}
```

合规代码

在此合规代码中，删除了上例中的强制类型转型，但这会导致原来的代码不合法，而且强调了一个潜在的问题： `B::g()` 根本不存在。本例的解决方案假设程序员拥有正确识别对象的动态类型为前提条件。

```

struct B {
  virtual ~B() = default;
};

struct D : B {
  virtual ~D() = default;
  virtual void g() { /* ... */ }
};

void f() {
  B *b = new D; // Corrected the dynamic object type.
```

```
  // ...
  void (D::*gptr)() = &D::g; // Moved static_cast to the next line.
  (static_cast<D *>(b)->*gptr)();
  delete b;
}
```

不合规代码

在此不合规代码中，指向成员的指针为空值，并且被传递作为指向成员的指针的表达式的第二个操作数，其结果行为未定义。

```
struct B {
  virtual ~B() = default;
};

struct D : B {
  virtual ~D() = default;
  virtual void g() { /* ... */ }
};

static void (D::*gptr)(); // Not explicitly initialized, defaults to nullptr.
void call_memptr(D *ptr) {
  (ptr->*gptr)();
}

void f() {
  D *d = new D;
  call_memptr(d);
  delete d;
}
```

合规代码

在此合规代码中， `gptr` 被初始化为一个合理的值，而不是默认的空指针。

```
struct B {
  virtual ~B() = default;
};

struct D : B {
```

```
  virtual ~D() = default;
  virtual void g() { /* ... */ }
};

static void (D::*gptr)() = &D::g; // Explicitly initialized.
void call_memptr(D *ptr) {
  (ptr->*gptr)();
}

void f() {
  D *d = new D;
  call_memptr(d);
  delete d;
}
```

参考文献

- Do not use pointer-to-member operators to access nonexistent members

OOP56-CPP. 遵循替换处理程序的规则

作者： 刘光聪10209986 评审人： 李永顺10110636

一般地，诸如 `new_handler`, `terminate_handler`, `unexpected_handler` 等处理程序能够自定义实现，并可以实现全局替换。例如，一个应用程序可以通过调用 `std::set_terminate` 设置一个自定义的终止处理程序，它会在程序终止时记录日志，方便后续的审计。
当置换任何的处理程序时，都需要满足合乎期望行为的语义需求。

新的处理程序

`new_handler` 的候选处理程序需求定义：

> *期望行为: `new_handler` 应该会执行如下某一动作：*
>
> - *让更多的内存可用，并且返回；*
> - *抛出类型为 `bad_alloc` ，或派生自 `bad_alloc` 的异常；*
> - *终止程序的执行，并且不返回至调用者。*

终止处理程序

`terminate_handler` 的候选处理程序需求定义：

> *期望行为: `terminate_handler` 应该终止程序执行，并且不返回至调用者。*

不可预期的处理程序

`unexpected_handler` 的候选处理程序需求定义：

> *期望行为： `unexpected_handler` 不应该返回*
> *其中， `unexpected_handler` 的特性已经被新的C++标准废弃。*

不合规代码

在此不合规代码中，一个 `new_handler` 的替代被设计出来。当动态内存管理器内存溢出时，用于释放可回收的资源。但是，这个例子并没有考虑一个特殊的场景，即所有的可回收的资源都已经被回收，但依然没有充足的内存满足内存分配的需求。此时，不会因抛出类型 `std::bad_alloc` 的异常而终止处理程序，或者终止程序的执行，并且没有返回给调用者，处理程序正常返回了。在低内存的场景下， `::operator new()` 的默认实现将陷入死循环。因为这样的场景在实际应用中极其少见，在测试环境中很可能发现不了这样的缺陷。

```
#include <new>

void custom_new_handler() {
  // Returns number of bytes freed.
  extern std::size_t reclaim_resources();
  reclaim_resources();
}

int main() {
  std::set_new_handler(custom_new_handler);

  // ...
}
```

合规代码

在此合规代码中， `custom_new_handler()` 使用了 `reclaim_resources()` 的返回值。如果它返回 `0` ，那么就没有足够的内存使得 `operator new` 成功。因此，按照替换 `handler` 的需求定义，可以抛出 `std::bad_alloc` 类型的异常。

```
#include <new>

void custom_new_handler() noexcept(false) {
  // Returns number of bytes freed.
  extern std::size_t reclaim_resources();
  if (0 == reclaim_resources()) {
    throw std::bad_alloc();
  }
}

int main() {
  std::set_new_handler(custom_new_handler);

  // ...
}
```

参考文献

- Honor replacement handler requirements

OOP57-CPP. 特殊成员函数和重载运算符优于C标准库函数

作者： 刘光聪10209986 评审人： 李永顺10110636

存在一些C标准库的函数用于对象的比特操作。例如， `std::memcmp` 用于按照比特顺序比较两个对象表示，而 `memcpy` 用于将对象表示的比特拷贝至目标缓存中。但是，对于一些类型，其行为是未定义的，或者行为异常。

不要使用 `std::memset` 初始化一个非平凡类的对象，因为这可能会导致非法地初始化对象的值表示。不要使用 `std::memcpy` (或者相关的比特操作的拷贝函数)拷贝初始化一个非平凡类的对象，因为这可能会导致非法地初始化副本的对象值表示。不要使用 `std::memcmp` (或者相关比特操作的比较函数)去比较非标准布局类的对象，因为这会导致对象表示的不合理的比较。所有情况，最好优先使用相应的替代方案。

C标准库函数	等价的C++功能
std::memcmp(), std::strcmp()	operator<(), operator>(), operator==(), 或 operator!=()

std::memcpy(), std::memmove(), std::strcpy()	拷贝构造函数 或operator=()
std::memset()	构造函数

不合规代码

在此不合规代码中，一个非平凡的类首先调用默认构造函数进行初始化，然后调用 `std::memset()` 重新初始化为默认状态，这将导致对象的不合理的重新初始化。

```
#include <cstring>
#include <iostream>

class C {
  int scalingFactor;
  int otherData;

public:
  C() : scalingFactor(1) {}

  void set_other_data(int i);
  int f(int i) {
    return i / scalingFactor;
  }
  // ...
};

void f() {
  C c;

  // ... Code that mutates c ...

  // Reinitialize c to its default state
  std::memset(&c, 0, sizeof(C));

  std::cout << c.f(100) << std::endl;
}
```

合规代码

在此合规代码中，使用 `clear` 替代 `std::memset` ；其中， `clear` 使用默认初始化，拷

贝-交换的操作实现。这个操作保证了对象被合理地初始化为默认状态，并且对象类型拥有一个优化了的赋值重载运算符，保证目标对象的成员变量不会被清除。

```
#include <iostream>
#include <utility>

class C {
  int scalingFactor;
  int otherData;

public:
  C() : scalingFactor(1) {}

  void set_other_data(int i);
  int f(int i) {
    return i / scalingFactor;
  }
  // ...
};

template <typename T>
T& clear(T &o) {
  using std::swap;
  T empty;
  swap(o, empty);
  return o;
}

void f() {
  C c;

  // ... Code that mutates c ...

  // Reinitialize c to its default state
  clear(c);

  std::cout << c.f(100) << std::endl;
}
```

不合规代码

在此不合规代码中，使用 `std::memcpy` 拷贝了一个非平凡的类 `C` 的对象。但是，每个对

象在 C::~C() 中试图删除 int* ，可能导致两次删除指针的错误，因为相同的指针被拷贝至 c2 中。

```
#include <cstring>

class C {
  int *i;

public:
  C() : i(nullptr) {}
  ~C() { delete i; }

  void set(int val) {
    if (i) { delete i; }
    i = new int{val};
  }

  // ...
};

void f(C &c1) {
  C c2;
  std::memcpy(&c2, &c1, sizeof(C));
}
```

合规代码

在合规代码中，C++定义了一个赋值重载运算符，并用于替代 std::memcpy 调用。

```
class C {
  int *i;
public:
  C() : i(nullptr) {}
  ~C() { delete i; }
  void set(int val) {
    if (i) { delete i; }
    i = new int{val};
  }
  C &operator=(const C &rhs) noexcept(false) {
    if (this != &rhs) {
      int *o = nullptr;
      if (rhs.i) {
```

```
        o = new int;
        *o = *rhs.i;
      }
      // Does not modify this unless allocation succeeds.
      delete i;
      i = o;
    }
    return *this;
  }
  // ...
};
void f(C &c1) {
  C c2 = c1;
}
```

不合规代码

在此不合规代码中， `std::memcmp()` 用于比较两个非标准布局的对象。因为 `std::memcmp()` 将按照比特位严格比较对象的表示，如果实现使用虚表(vtable)指针，并作为对象表示的一部分， `std::memcmp()` 将比较虚表(vtable)指针。

```
#include <cstring>

class C {
  int i;

public:
  virtual void f();

  // ...
};

void f(C &c1, C &c2) {
  if (!std::memcmp(&c1, &c2, sizeof(C))) {
    // ...
  }
}
```

合规代码

在此合规代码中，C++定义了一个相等性比较的运算符重载，用于替代 `std::memcmp` 的

调用。

```
class C {
  int i;

public:
  virtual void f();

  bool operator==(const C &rhs) const {
    return rhs.i == i;
  }

  // ...
};

void f(C &c1, C &c2) {
  if (c1 == c2) {
    // ...
  }
}
```

OOP58-CPP. 拷贝操作不允许修改原对象

作者： 刘光聪10209986 评审人： 李永顺10110636

拷贝操作(拷贝构造函数和拷贝赋值运算符重载)用于从原对象拷贝特征属性到目标对象，其目标对象成为原对象的一个副本。在理想的情况下，拷贝操作应该有一个惯用的签名。拷贝构造函数，即 `T(const T&)` ；及其拷贝赋值运算符重载，即 `T& operator=(const T&)` ； 如果拷贝构造函数和拷贝赋值重载运算符不使用惯用的签名，则不符合的 `CopyAssignable` 概念的要求，如此将导致类型与标准库通用函数功能的不兼容。

当实现一个拷贝操作时，对于任何源对象的操作数或全局信息，不要改变任何外部可观察的成员。外部可观察成员包括但不限于参与比较或相等操作的成员，通过公共 `API` 公开其值的成员，及其全局变量。

在C11之前，复制操作改变源对象的唯一途径就是提供移动语义。然而，语言并没有提供一种有效的方法，当源对象的声明周期结束时来执行此操作，从而导致诸如 `std::auto_ptr` 脆弱的API设计。在C11之后，对于这样的情况，更合理的候选方案就是使用移动操作替代拷贝操作。

auto_ptr

例如，在C++03中， `std::auto_ptr` 拥有如下的拷贝操作签名:
函数	签名
————	————
拷贝构造函数	auto_ptr(auto_ptr &A);
拷贝赋值运算符重载	auto_ptr& operator=(auto_ptr &A);
无论是拷贝构造函数，还是拷贝赋值重载运算符，通过调用 `this->reset(A.release())` 都会修改原对象 `A` 。但是，这使得诸如 `std::sort` 等标准库算法成为不可用，因为这些算法不仅要拷贝原对象，然后再使用原对象做比较。考虑如下 `std::sort` 实现快速排序算法的实现。

```
// ...
value_type pivot_element = *mid_point;
// ...
```

在这里，排序算法假定 `pivot_element` 和 `*mid_point` 拥有等价的值表示，并且能够相互比较相等性。但是，对于 `std::auto_ptr` ，违背了这个假设；因为 `*mid_point` 已经被修改，并且其结果行为未定义。

在C11中，引入了新的智能指针类型 `std::unique_ptr` ，用于替代 `std::auto_ptr` 。它能够更好地定义对象所有权的语义。 `std::unique_ptr` 并没有通过拷贝操作修改原对象，而是通过显式地删除了拷贝构造函数及其拷贝赋值运算符重载，并使用移动构造函数及其移动赋值运算符重载代替。最终， `std::auto_ptr` 被C11废弃。

不合规代码

在此不合规代码中， `A` 的拷贝操作通过将原对象的成员变量置位为 `0` ，以此修改了原对象的状态。当调用 `std::fill()` 时，第一个元素拷贝了 `obj` 的值，此时 `obj.m` 值为12，然后被置位为 `0` 。随后的9份拷贝中，其值都为 `0` 。

```
#include <algorithm>
#include <vector>

class A {
  mutable int m;

public:
```

```
  A() : m(0) {}
  explicit A(int m) : m(m) {}

  A(const A &other) : m(other.m) {
    other.m = 0;
  }

  A& operator=(const A &other) {
    if (&other != this) {
      m = other.m;
      other.m = 0;
    }
    return *this;
  }

  int get_m() const { return m; }
};

void f() {
  std::vector<A> v{10};
  A obj(12);
  std::fill(v.begin(), v.end(), obj);
}
```

合规代码

在此合规代码中，对 `A` 的拷贝操作并未修改原对象，因此保证了 `vector` 包含了 `obj` 等价的多份拷贝。与此相反， `A` 也提供了移动操作，以便安全地修改原对象。

```
#include <algorithm>
#include <vector>

class A {
  int m;

public:
  A() : m(0) {}
  explicit A(int m) : m(m) {}

  A(const A &other) : m(other.m) {}
  A(A &&other) : m(other.m) { other.m = 0; }
```

```
  A& operator=(const A &other) {
    if (&other != this) {
      m = other.m;
    }
    return *this;
  }

  A& operator=(A &&other) {
    m = other.m;
    other.m = 0;
    return *this;
  }

  int get_m() const { return m; }
};

void f() {
  std::vector<A> v{10};
  A obj(12);
  std::fill(v.begin(), v.end(), obj);
}
```

Rule 10. 并发

CON50-CPP. 不要销毁一个正在被锁定的互斥对象

作者：赵庆轩10146152 评审人： 曹笑10116982

互斥对象用于保护共享数据的并发访问。如果一个互斥对象在一个线程等待这个互斥锁的时候被销毁，则临界区和共享数据不再是受保护的。

C++标准, [thread.mutex.class], 段落5 [ISO/IEC 14882-2014], 有如下描述：

> *如果程序销毁了一个仍被任何其它线程持有的互斥对象，或者拥有互斥对象的线程终止了，则该程序的行为是未定义的。*
> *类似的文字也存在于 `std::recursive_mutex` ， `std::timed_mutex` ， `std::recursive_timed_mutex` ，和 `std::shared_timed_mutex` 。这些描述意味着，当一个线程正在等待的互斥对象被破坏时，这会造成未定义行为。*

不合规代码

下面这个不合规代码中，创建了调用do_work()函数的多个线程，每个线程传入一个唯一的数字作为ID。
不幸的是，此代码包含一个竞争条件，这会导致互斥对象在一个线程拥有它时被破坏，因为函数start_threads()可能先于线程 `do_work()` 结束，互斥对象已经或者正在被析构。

```
#include <mutex>
#include <thread>

const size_t maxThreads = 10;

void do_work(size_t i, std::mutex *pm) {
  std::lock_guard<std::mutex> lk(*pm);

  // Access data protected by the lock.
}

void start_threads() {
  std::thread threads[maxThreads];
  std::mutex m;

  for (size_t i = 0; i < maxThreads; ++i) {
    threads[i] = std::thread(do_work, i, &m);
  }
}
```

合规代码

下面这个合规代码通过延长互斥对象的生命周期来消除竞争条件。

```
#include <mutex>
#include <thread>

const size_t maxThreads = 10;

void do_work(size_t i, std::mutex *pm) {
  std::lock_guard<std::mutex> lk(*pm);

  // Access data protected by the lock.
}
```

```
std::mutex m;

void start_threads() {
  std::thread threads[maxThreads];

  for (size_t i = 0; i < maxThreads; ++i) {
    threads[i] = std::thread(do_work, i, &m);
  }
}
```

合规代码

下面这个合规代码通过在互斥对象的析构方法被调用前通过线程同步join()来消除竞争条件

```
#include <mutex>
#include <thread>

const size_t maxThreads = 10;

void do_work(size_t i, std::mutex *pm) {
  std::lock_guard<std::mutex> lk(*pm);

  // Access data protected by the lock.
}
void run_threads() {
  std::thread threads[maxThreads];
  std::mutex m;

  for (size_t i = 0; i < maxThreads; ++i) {
    threads[i] = std::thread(do_work, i, &m);
  }

  for (size_t i = 0; i < maxThreads; ++i) {
    threads[i].join();
  }
}
```

参考文献

- CON50-CPP. Do not destroy a mutex while it is locked

CON51-CPP. 确保异常条件下持有的锁被释放

作者：赵庆轩10146152 评审人： 曹笑10116982

互斥对象用来保护对共享数据的访问，通过lock()成员函数加锁，通过unlock()成员函数解锁。如果在调用lock()和unlock()之间发生了异常，那么异常就会改变控制流，导致unlock()没有被调用，互斥对象保持在锁定状态，被互斥对象保护的临界区就无法执行，这很有可能导致死锁。

抛出异常不允许使互斥对象一直保持被锁定状态。如果一个互斥对象被锁住，并且被该互斥对象保护的临界区出现了一个异常，那么在重新抛出异常或继续执行直到子控制流解锁了这个互斥对象前必须把互斥对象解锁作为异常处理的一部分。

C++提供的锁类lock_guard, unique_lock和 shared_lock可以用一个互斥对象来初始化。在它们的构造函数中，锁对象锁住了互斥对象，在析构函数中，锁对象解锁互斥对象。lock_guard 类提供了一个对互斥对象的简单RAII封装。unique_lock 和shared_lock也使用RAII提供额外的功能，比如手工控制锁策略。unique_lock类防止锁被复制，但允许锁的所有权被转移到另外一个锁。shared_lock类允许互斥对象被几个锁共享。对上述三个类，如果异常发生，可以将控制流从锁的范围中取出，析构方法会把互斥对象解锁，程序能继续正常执行。应该首选使用这些锁对象来保证异常抛出时互斥对象被正确地释放。

不合规代码

下面这个不合规代码中，操作共享数据时，通过锁住互斥对象来保护临界区。当操作完成后，解锁互斥对象。但是，如果在操作共享数据的时候异常发生了，那么互斥对象就会保持锁定状态。

```
#include <mutex>

void manipulate_shared_data(std::mutex &pm) {
  pm.lock();

  // Perform work on shared data.

  pm.unlock();
}
```

合规代码（手工解锁）

下面的合规代码在对共享数据操作时捕获了所有抛出的异常，然后在重新抛出异常前解锁

了互斥对象。

```
#include <mutex>

void manipulate_shared_data(std::mutex &pm) {
  pm.lock();
  try {
    // Perform work on shared data.
  } catch (...) {
    pm.unlock();
    throw;
  }
  pm.unlock(); // in case no exceptions occur
}
```

合规代码（锁住对象）

下面的合规代码使用lock_guard对象来保证互斥对象即使在异常发生时也不被锁住，而不需要依赖异常处理机制和手工资源管理。

```
#include <mutex>

void manipulate_shared_data(std::mutex &pm) {
  std::lock_guard<std::mutex> lk(pm);

  // Perform work on shared data.
}
```

参考文献

- CON51-CPP. Ensure actively held locks are released on exceptional conditions

CON52-CPP. 多个线程访问位字段时需要防止数据竞争

作者：赵庆轩10146152 评审人： 曹笑10116982

访问位字段时，线程可能会无意中访问相邻内存中的位字段。这是因为编译器需要在一个存储单元中存储多个相邻的位字段。因此，数据竞争不仅存在于由多个线程访问的位域，而且还存在于共享同一个byte或word的其他位域上。这个问题很难诊断，因为相同的内存位置可能被多个线程同时修改。

防止并发编程中数据竞争的一种方法是使用互斥锁。当有多个线程时，互斥锁保证所有线程安全地访问共享对象。然而，互斥锁不能控制其他未加锁的线程访问共享资源。不幸的是，没有合适的方法来确定哪些相邻的位字段可以与所需的位字段一起存储。

另一种方法是在任何两个位字段之间插入非位字段成员，以确保每个位字段在其存储单元内是唯一。这种技术有效地保证没有两个位字段需要同时访问。

不合规代码(位域)

相邻位字段可能存储在单个存储器位置。因此，在不同的线程中修改相邻的位域可能导致不可预期的结果。如下面代码所示。

```
struct MultiThreadedFlags {
  unsigned int flag1 : 2;
  unsigned int flag2 : 2;
};

MultiThreadedFlags flags;

void thread1() {
  flags.flag1 = 1;
}

void thread2() {
  flags.flag2 = 2;
}
```

例如，可能是如下的访问序列。

```
Thread 1: register 0 = flags
Thread 1: register 0 &= ~mask(flag1)
Thread 2: register 0 = flags
Thread 2: register 0 &= ~mask(flag2)
Thread 1: register 0 |= 1 << shift(flag1)
Thread 1: flags = register 0
Thread 2: register 0 |= 2 << shift(flag2)
Thread 2: flags = register 0
```

合规代码(位域，C++11及以后，互斥对象)

下面的合规代码通过互斥锁来防止数据竞争。

```
#include <mutex>

struct MultiThreadedFlags {
  unsigned int flag1 : 2;
  unsigned int flag2 : 2;
};

struct MtfMutex {
  MultiThreadedFlags s;
  std::mutex mutex;
};

MtfMutex flags;

void thread1() {
  std::lock_guard<std::mutex> lk(flags.mutex);
  flags.s.flag1 = 1;
}

void thread2() {
  std::lock_guard<std::mutex> lk(flags.mutex);
  flags.s.flag2 = 2;
}
```

合规代码(C++11)

在下面这个合规代码中，两个线程同时修改两个不同的非位字段成员。由于成员在内存中占用不同的字节，所以不需要并发保护。

```
struct MultiThreadedFlags {
  unsigned char flag1;
  unsigned char flag2;
};

MultiThreadedFlags flags;

void thread1() {
  flags.flag1 = 1;
}

void thread2() {
  flags.flag2 = 2;
```

```
}
```

不像早期版本，C++ 11和之后版本显式定义一个内存位置并提供以下说明，段落4[ISO/IEC 14882-2014]:

> *一个位字段和一个相邻的非位字段位于不同的内存位置，因此两个线程可以互不干扰地更新。如果两个字段在一个嵌套的结构定义，或者被两个零长度的位域声明分开，或由一个非位域声明分离，同样适用上述原则。但如果这两个位字段之间的所有成员也是非零宽度的位域，则同时更新同一结构中两个位字段是不安全的。*

几乎可以肯定的是，flag1和flag2存储在同一个字节中。在使用早期版本的编译器时，如果该字节被两个线程交叉访问，可能只有一个标记被设置，另外一个仍然是之前的值，因为处理器最小操作单位是字节。C++ 11标准修订之前，无法保证这些共享存储的位域标记同时被修改。

参考文献

- CON52-CPP. Prevent data races when accessing bit-fields from multiple threads

CON53-CPP. 按预定义的顺序加锁来避免死锁

作者：赵庆轩10146152 评审人： 曹笑10116982

互斥锁用于防止多个线程同时访问同一共享资源造成的数据竞争。当多个线程同时持有对方的锁时，会造成程序死锁。死锁发生需要下面四个条件：

1. 互斥条件：一个资源每次只能被一个线程使用。
2. 请求与保持条件：一个线程因请求资源而阻塞时，对已获得的资源保持不放。
3. 非抢占条件：线程已获得的资源，在未使用完之前，不能强行抢占。
4. 循环等待条件：若干线程之间形成一种头尾相接的循环等待资源关系。

死锁需要所有四个条件，所以防止死锁的方法就是阻止四个条件同时发生。一个简单的解决办法是为互斥锁设置一个预定义的顺序，以防止循环等待。

不合规代码

下面这个代码的运行效果在不同的运行环境和平台有所不同。在函数deposit()中，如果线程1尝试对ba2加互斥锁的同时，线程2也在试图锁定ba1时就会发生死锁。

```
#include <mutex>
#include <thread>

class BankAccount {
  int balance;
public:
  std::mutex balanceMutex;
  BankAccount() = delete;
  explicit BankAccount(int initialAmount) : balance(initialAmount) {}
  int get_balance() const { return balance; }
  void set_balance(int amount) { balance = amount; }
};

int deposit(BankAccount *from, BankAccount *to, int amount) {
  std::lock_guard<std::mutex> from_lock(from->balanceMutex);

  // Not enough balance to transfer.
  if (from->get_balance() < amount) {
    return -1; // Indicate error
  }
  std::lock_guard<std::mutex> to_lock(to->balanceMutex);

  from->set_balance(from->get_balance() - amount);
  to->set_balance(to->get_balance() + amount);

  return 0;
}

void f(BankAccount *ba1, BankAccount *ba2) {
  // Perform the deposits.
  std::thread thr1(deposit, ba1, ba2, 100);
  std::thread thr2(deposit, ba2, ba1, 100);
  thr1.join();
  thr2.join();
}
```

合规代码

下面的合规代码通过预定义的顺序在deposit()中循环等待。每个线程都使用预分配的BankAccount ID锁定。

```
#include <atomic>
```

```
#include <mutex>
#include <thread>

class BankAccount {
  static std::atomic<unsigned int> globalId;
  const unsigned int id;
  int balance;
public:
  std::mutex balanceMutex;
  BankAccount() = delete;
  explicit BankAccount(int initialAmount) : id(globalId++), balance(initialAmount) {
}
  unsigned int get_id() const { return id; }
  int get_balance() const { return balance; }
  void set_balance(int amount) { balance = amount; }
};

std::atomic<unsigned int> BankAccount::globalId(1);

int deposit(BankAccount *from, BankAccount *to, int amount) {
  std::mutex *first;
  std::mutex *second;

  if (from->get_id() == to->get_id()) {
    return -1; // Indicate error
  }

  // Ensure proper ordering for locking.
  if (from->get_id() < to->get_id()) {
    first = &from->balanceMutex;
    second = &to->balanceMutex;
  } else {
    first = &to->balanceMutex;
    second = &from->balanceMutex;
  }
  std::lock_guard<std::mutex> firstLock(*first);
  std::lock_guard<std::mutex> secondLock(*second);

  // Check for enough balance to transfer.
  if (from->get_balance() >= amount) {
    from->set_balance(from->get_balance() - amount);
    to->set_balance(to->get_balance() + amount);
    return 0;
```

```
  }
  return -1;
}

void f(BankAccount *ba1, BankAccount *ba2) {
  // Perform the deposits.
  std::thread thr1(deposit, ba1, ba2, 100);
  std::thread thr2(deposit, ba2, ba1, 100);
  thr1.join();
  thr2.join();
}
```

合规代码

下面的合规代码使用标准模板库，以确保不会因为循环等待而发生死锁。std::lock()函数接受一个可锁定的对象和锁[ISO/IEC 14882-2014]。典型的应用是使用lock()，try_lock()，和unlock()组合，来试图锁定对象，在未获取到资源时回退。

```
#include <mutex>
#include <thread>

class BankAccount {
  int balance;
public:
  std::mutex balanceMutex;
  BankAccount() = delete;
  explicit BankAccount(int initialAmount) : balance(initialAmount) {}
  int get_balance() const { return balance; }
  void set_balance(int amount) { balance = amount; }
};

int deposit(BankAccount *from, BankAccount *to, int amount) {
  // Create lock objects but defer locking them until later.
  std::unique_lock<std::mutex> lk1(from->balanceMutex, std::defer_lock);
  std::unique_lock<std::mutex> lk2(to->balanceMutex, std::defer_lock);

  // Lock both of the lock objects simultaneously.
  std::lock(lk1, lk2);

  if (from->get_balance() >= amount) {
    from->set_balance(from->get_balance() - amount);
    to->set_balance(to->get_balance() + amount);
```

```
    return 0;
  }
  return -1;
}

void f(BankAccount *ba1, BankAccount *ba2) {
  // Perform the deposits.
  std::thread thr1(deposit, ba1, ba2, 100);
  std::thread thr2(deposit, ba2, ba1, 100);
  thr1.join();
  thr2.join();
}
```

参考文献

- CON53-CPP. Avoid deadlock by locking in a predefined order

CON54-CPP. 在循环中封装那些可能产生伪激活的函数

作者：赵庆轩10146152 评审人： 曹笑10116982

std::condition_variable 类的wait(), wait_for()和 wait_until()成员函数通过暂时放弃一个占有的互斥对象，使得其它可能请求这个互斥对象的线程可以继续。这些函数必须总是在通过一个互斥对象锁定来保护的代码段里被调用。通常情况下，作为其他线程调用notify_one()或notify_all()成员函数的请求结果，这些等待线程只有在被通知时才会继续执行。

wait()函数必须在检查一个条件判定是否满足的循环语句中被调用。条件判定语句是一个函数变量构造的表达式，必须为true时才允许继续执行。线程在执行wait(), wait_for(), wait_until()或者其它机制时暂停执行，并假定在稍后当条件语句为true且线程被通知时恢复执行。

```
#include <condition_variable>
#include <mutex>

extern bool until_finish(void);
extern std::mutex m;
extern std::condition_variable condition;

void func(void) {
  std::unique_lock<std::mutex> lk(m);
```

```
  while (until_finish()) {  // Predicate does not hold.
    condition.wait(lk);
  }

  // Resume when condition holds.
}
```

通知机制通知等待线程并允许它检查其条件判定。在另外一个线程中调用notify_all()并不能决定哪个等待线程被激活。条件判定允许被通知的线程在收到通知时来决定它们是否应该恢复执行。

不合规代码

下面这个不合规代码监视一个链表，当链表非空时，分配一个线程来消费列表中的元素。

该线程通过使用 wait()暂停执行，当收到通知并假定列表中有可以消费的元素时恢复执行。如果通知线程使用notify_all()来通知所有线程，即使列表是空的，线程也有可能被通知。使用notify_all()而不是notify_one()来完成通知会更为常见。（参考 CON55-CPP. 使用条件变量时保持线程的安全性和活性 获取更多信息。）
条件判定通常是循环中条件表达式的否定。在下面这个不合规代码示例中，从一个链表中移除一个元素的条件判定是(list->next != nullptr)，而while循环的条件表达式是(list->next == nullptr)。

这个不合规代码示例把对wait()调用放到if语句块内，因此在收到通知时无法检查条件判定。如果通知是虚假的或恶意的，线程会被提前激活。

```
#include <condition_variable>
#include <mutex>

struct Node {
  void *node;
  struct Node *next;
};

static Node list;
static std::mutex m;
static std::condition_variable condition;

void consume_list_element(std::condition_variable &condition) {
  std::unique_lock<std::mutex> lk(m);

  if (list.next == nullptr) {
```

```
    condition.wait(lk);
  }

  // Proceed when condition holds.
}
```

合规代码（带谓词的显式循环）

下面这个合规代码在一个while循环内调用 wait()成员函数，不管是调用前还是调用后都检查这个条件。

```
#include <condition_variable>
#include <mutex>

struct Node {
  void *node;
  struct Node *next;
};

static Node list;
static std::mutex m;
static std::condition_variable condition;

void consume_list_element() {
  std::unique_lock<std::mutex> lk(m);

  while (list.next == nullptr) {
    condition.wait(lk);
  }

  // Proceed when condition holds.
}
```

合规代码（带lambda表达式的隐式循环）

std::condition_variable::wait()函数有一个重载的形式，接受一个表示断言的函数对象。这种形式的 wait()实现起来就像while (!pred()) wait(lock);下面的合规代码使用lambda表达式作为一个断言并传入到wait()函数。当可以安全执行时，这个断言期望返回true，它逆转了那个在显式loop循环中使用的断言逻辑。

```
#include <condition_variable>
```

```
#include <mutex>

struct Node {
  void *node;
  struct Node *next;
};

static Node list;
static std::mutex m;
static std::condition_variable condition;

void consume_list_element() {
  std::unique_lock<std::mutex> lk(m);

  condition.wait(lk, []{ return list.next; });
  // Proceed when condition holds.
}
```

参考文献

- CON54-CPP. Wrap functions that can spuriously wake up in a loop

CON55-CPP. 使用条件变量时保持线程的安全性和活性

作者：赵庆轩10146152 评审人： 曹笑10116982

使用条件变量时需要考虑线程安全性和活性。线程安全属性要求，在多线程环境中所有对象保持一致的状态[Lea 2000]。线程活跃性要求每个操作或函数调用没有中断，例如，没有死锁。
条件变量必须在while循环中使用。（见CON54-CPP.在循环中封装那些可能产生伪激活的函数）。为了保证活性，程序必须在调用成员函数condition_variable::wait()之前测试while循环条件。前期测试用来检查是否另一线程已满足条件判定并已发送通知。通知发送后再调用wait()，则会使线程处于无限阻塞中。

为了保证线程安全，从wait()返回时程序必须测试循环条件。当一个线程调用wait()，它将处于阻塞状态直到其条件变量被condition_variable::notify_all() 或 condition_variable::notify_one()函数调用激活。

调用成员函数notify_one()时会唤醒一个阻塞状态的线程。如果多个线程在相同的条件变量上等待，调度器可以选择要唤醒的任何线程（假设所有线程具有相同的优先级）。

调用成员函数notify_all()可以唤醒所有阻塞状态的线程。线程的激活顺序是不确定的。因此，一个不相关的线程，即使它应该保持休眠状态，也可能开始执行，当这个线程发现它的条件判断得到满足时，就会恢复执行。

由于这些原因，线程必须在wait()函数返回时检查条件判断。在调用wait()之前和之后，使用一个while循环检查条件判断是个不错的选择。

如果每个线程使用一个独特的条件变量，使用notify_one()是比较安全的。如果多个线程共享一个条件变量，只有在满足以下条件时，notify_one()的使用才是安全的：

1. 所有线程必须醒来后执行相同的操作集合，这意味着每次调用notify_one()时，任何一个线程都可以被唤醒。
2. 只需要唤醒一个线程。

如果使用notify_one()唤醒线程不安全，则可以使用notify_all()函数唤醒所有的阻塞线程。

不合规代码(notify_one())

下面这个不合规代码示例使用五个线程，每个进程按照创建时的顺序执行。currentStep 变量保存当前步长等级并在相应线程执行后递增。最后，另一个线程被分配执行。每一个线程在其步长变量准备好之前等待，并且wait()调用被封装在一个while循环中，符合标准CON54-CPP.在循环中封装那些可能产生伪激活的函数。

```
#include <condition_variable>
#include <iostream>
#include <mutex>
#include <thread>

std::mutex mutex;
std::condition_variable cond;

void run_step(size_t myStep) {
  static size_t currentStep = 0;
  std::unique_lock<std::mutex> lk(mutex);

  std::cout << "Thread " << myStep << " has the lock" << std::endl;

  while (currentStep != myStep) {
    std::cout << "Thread " << myStep << " is sleeping..." << std::endl;
    cond.wait(lk);
    std::cout << "Thread " << myStep << " woke up" << std::endl;
  }
```

```
  // Do processing...
  std::cout << "Thread " << myStep << " is processing..." << std::endl;
  currentStep++;

  // Signal awaiting task.
  cond.notify_one();

  std::cout << "Thread " << myStep << " is exiting..." << std::endl;
}

int main() {
  constexpr size_t numThreads = 5;
  std::thread threads[numThreads];

  // Create threads.
  for (size_t i = 0; i < numThreads; ++i) {
    threads[i] = std::thread(run_step, i);
  }

  // Wait for all threads to complete.
  for (size_t i = numThreads; i != 0; --i) {
    threads[i - 1].join();
  }
}
```

在本例中，所有线程共享一个条件变量。每个线程都有自己的独特条件判断，因为每个线程执行之前需要不同的currentStep值。当条件变量发出信号时，任何等待线程都可以唤醒。下表说明了一个可能的执行情况，影响了线程的活性。如果碰巧通知的线程不是具有下一步值的线程，则该线程将再次等待。系统中没有其它的通知，最终会耗尽可用线程池。

时间	线程	当前步长	执行
6	—	—	没有运行的线程，需要一个条件变量来唤醒其他线程。
5	3	2	唤醒线程3（调度程序选择）：条件判断是false，wait()
4	1	1	第一次执行线程1：条件判断是true，currentStep++； notify_one()

3	0	0	第一次执行线程0：条件判断是true，currentStep++；notify_one()
2	4	0	第一次执行线程4：条件判断是false，wait()
1	2	0	第一次执行线程2：条件判断是false，wait()
0	3	0	第一次执行线程3：条件判断是false，wait()

该示例不符合线程的活性属性。

合规代码(notify_all())

下面这个合规代码使用notify_all()通知所有等待的线程，而不是每次随机通知一个线程。只修改了上面示例中的run_step() 方法。

```
#include <condition_variable>
#include <iostream>
#include <mutex>
#include <thread>

std::mutex mutex;
std::condition_variable cond;

void run_step(size_t myStep) {
  static size_t currentStep = 0;
  std::unique_lock<std::mutex> lk(mutex);

  std::cout << "Thread " << myStep << " has the lock" << std::endl;

  while (currentStep != myStep) {
    std::cout << "Thread " << myStep << " is sleeping..." << std::endl;
    cond.wait(lk);
    std::cout << "Thread " << myStep << " woke up" << std::endl;
  }

  // Do processing ...
  std::cout << "Thread " << myStep << " is processing..." << std::endl;
  currentStep++;

  // Signal ALL waiting tasks.
  cond.notify_all();

  std::cout << "Thread " << myStep << " is exiting..." << std::endl;
```

```
}

// ... main() unchanged ...
```

唤醒所有线程保证线程活性，因为每个线程执行条件判断测试，一定会有一个线程被成功唤醒并执行。

合规代码(在每线程一个唯一条件变量的情况下使用notify_one())

另一个合规代码是每个线程使用一个唯一的条件变量（都使用同一个互斥锁）。在这种情况下，notify_one()只唤醒等待的线程。这种方案比使用notify_all()更有效，因为只有所需的线程被唤醒。
接收到信号的线程，其条件判断必须是true，否则会发生死锁。

```
#include <condition_variable>
#include <iostream>
#include <mutex>
#include <thread>

constexpr size_t numThreads = 5;

std::mutex mutex;
std::condition_variable cond[numThreads];

void run_step(size_t myStep) {
  static size_t currentStep = 0;
  std::unique_lock<std::mutex> lk(mutex);

  std::cout << "Thread " << myStep << " has the lock" << std::endl;

  while (currentStep != myStep) {
    std::cout << "Thread " << myStep << " is sleeping..." << std::endl;
    cond[myStep].wait(lk);
    std::cout << "Thread " << myStep << " woke up" << std::endl;
  }

  // Do processing ...
  std::cout << "Thread " << myStep << " is processing..." << std::endl;
  currentStep++;
```

```
  // Signal next step thread.
  if ((myStep + 1) < numThreads) {
    cond[myStep + 1].notify_one();
  }

  std::cout << "Thread " << myStep << " is exiting..." << std::endl;
}

// ... main() unchanged ...
```

参考文献

- CON55-CPP. Preserve thread safety and liveness when using condition variables

CON56-CPP. 不要刻意地锁一个已经被其它线程持有的非递归锁

作者：赵庆轩10146152 评审人： 曹笑10116982

C++标准库提供递归和非递归互斥类用来保护临界区。递归互斥类（std::recursive_mutex 和 std::recursive_timed_mutex）与非递归互斥类（std::mutex，std::timed_mutex，和 std::shared_timed_mutex）不同，递归互斥锁可以由当前拥有互斥锁的线程递归锁定。所有的互斥类可以使用try_lock()，try_lock_for()，try_lock_until()，try_lock_shared_for()和try_lock_shared_until()等方法投机地锁定。这些函数的调用线程试图获取互斥锁的所有权，但是获取不到也不会阻塞线程。相反，它返回一个布尔值，指定是否取得互斥锁的所有权。
C++标准，[thread.mutex.requirements.mutex]，段落14和15 [ISO/IEC 14882-2014]，说明如下：

> *表达式m.try_lock()应该是结构良好的，且具有以下语义：*
> *要求：如果m是 std::mutex, std::timed_mutex, 或 std::shared_timed_mutex类型，调用线程不会拥有互斥锁。*

此外，[thread.timedmutex.class]，段落3，部分规定如下：

> *一个拥有timed_mutex对象的线程在这个对象上调用lock()，try_lock()，try_lock_for()，或try_lock_until()时，程序的结果是不可预期的。*

最后[thread.sharedtimedmutex.class]，段落3，部分规定如下：

> *如果一个拥有shared_timed_mutex锁的线程尝试递归加锁时，程序的结果也是不可预期的。*

因此，试图递归锁定一个已经加锁的非递归互斥对象，其结果是不可预期的。不要在一个已经拥有互斥锁的非递归互斥对象上调用try_lock()，try_lock_for()，try_lock_until()，try_lock_shared_for()，或try_lock_shared_until()。

不合规代码

下面这个不合规代码示例中，互斥对象m被线程的初始入口点锁住了，并且被在同一线程中的do_work()函数刻意地加锁，由于不是一个递归的互斥对象，导致了未定义行为。在大多数应用中，这可能导致死锁。

```
#include <mutex>
#include <thread>

std::mutex m;

void do_thread_safe_work();

void do_work() {
  while (!m.try_lock()) {
    // The lock is not owned yet, do other work while waiting.
    do_thread_safe_work();
  }
  try {

    // The mutex is now locked; perform work on shared resources.
    // ...

  // Release the mutex.
  catch (...) {
    m.unlock();
    throw;
  }
  m.unlock();
}

void start_func() {
  std::lock_guard<std::mutex> lock(m);
  do_work();
}
```

```
int main() {
  std::thread t(start_func);

  do_work();

  t.join();
}
```

合规代码

下面的合规代码移除了线程初始入口点的锁，允许互斥对象被刻意的锁住，但不是递归的。

```
#include <mutex>
#include <thread>

std::mutex m;

void do_thread_safe_work();

void do_work() {
  while (!m.try_lock()) {
    // The lock is not owned yet, do other work while waiting.
    do_thread_safe_work();
  }
  try {
    // The mutex is now locked; perform work on shared resources.
    // ...

  // Release the mutex.
  catch (...) {
    m.unlock();
    throw;
  }
  m.unlock();
}

void start_func() {
  do_work();
}
```

```
int main() {
  std::thread t(start_func);

  do_work();

  t.join();
}
```

参考文献

- CON56-CPP. Do not speculatively lock a non-recursive mutex that is already owned by the calling thread

Rule 11. 杂项

MSC50-CPP. 禁止使用 std::rand() 生成伪随机数

作者：曹笑10116982 评审人：赵庆轩10146152

伪随机数生成器使用数值算法生成一系列具有良好统计属性的随机值，但是其生成的数字并非真正随机。

C++标准库函数 `std::rand()` ，并没有保证随机序列生成的质量。 `std::rand()` 的某种实现生成的随机值拥有相对较短的周期，并且其值可预期。应用程序对伪随机数的需求非常强烈，期望应用伪随机数生成器满足他们的需求。

不合规代码

在该代码中，通过调用 `rand()` 函数随机生成一个数字表示的 `ID` 。但是，随机生成的 `ID` 序列是可预测的，并具有有限的随机性。此外根据 `RAND_MAX` 的值，得到的结果可能存在模偏差。

```
#include <cstdlib>
#include <string>

void f() {
  std::string id("ID"); // Holds the ID, starting with the characters "ID" followed
                        // by a random integer in the range [0-10000].
  id += std::to_string(std::rand() % 10000);
```

```
  // ...
}
```

合规代码

C++标准库提供了一个更好的生成伪随机数的机制。它将随机数生成分为两个部分：一是算法(引擎)，负责生成随机值。二是密度函数(分布函数)，它负责描述随机值的概率分布。概率分布对象并非严格必须的，但是其保证了随机值分布在一个给定的范围内，而不是因为存在较大方差生成一个不合理的随机值分布。在代码中，使用了「Mersenne Twister」算法来用于生成随机数，并且应用正态分布，并且取得负模偏差。

```
#include <random>
#include <string>

void f() {
  std::string id("ID"); // Holds the ID, starting with the characters "ID" followed
                        // by a random integer in the range [0-10000].
  std::uniform_int_distribution<int> distribution(0, 10000);
  std::random_device rd;
  std::mt19937 engine(rd());
  id += std::to_string(distribution(engine));
  // ...
}
```

在合规代码中，给随机数生成引擎设置了种子，同时其实现遵循了规则MSC51-CPP，保证随机数生成器拥有合理的种子。

参考文献

- MSC50-CPP Do not use std::rand() for generating pseudorandom numbers

MSC51-CPP. 确保随机数生成引擎使用合适的种子

作者：曹笑10116982 评审人：赵庆轩10146152

PRNG(伪随机数生成器)是一种非确定性的算法，它能够生成近似随机数属性的数字序列。每个序列由PRNG的初始状态，及其改变状态的算法完全确定。大多数PRNG会尽最大可能地设置初始状态，这个状态也常称为「种子状态」。因此，设置初始状态也常被称为给PRNG播种。

当它就在相同的初始状态下调用，或者没有显式地给它播种，或者播的种子是一个常量值，运行程序多次，将得到相同的随机数序列。考虑一个PRNG函数，它被设置了某一初始的种子，然后调用它生成一个随机值的序列。如果这个PRNG随后被使用相同的初始种子，它将生成相同的随机序列。

结果，当一个有缺陷的，且已播种的PRNG首次运行之后，攻击者便可能在未来运行中预知生成的随机序列。不恰当的种子，或者给PRNG播种失败，都可能导致安全漏洞，尤其在安全协议实现中出现该问题。解决方案就是保证使用不可预知的，或者不能被攻击者控制的初始种子的值给PRNG播种。一个合理的，且已播种的PRNG每次运行将会产生不同随机数序列。

但是，并非所有随机数生成器都能够播种。真随机数生成器依赖于生成完全不可预知的结果的硬件设备，它不用或不能被播种。一些高质量的PRNG，例如在某些 `UNIX` 系统上的 `/dev/random` 设备，也不能播种。这条规则进适用于能够播种的，算法型的PRNG。

不合规代码

在代码中，使用「Mersenne Twister」引擎生成10个伪随机数的序列。无论该代码执行多少次，它总是生成相同的序列，因为引擎使用默认的种子。

```
#include <random>
#include <iostream>

void f() {
  std::mt19937 engine;

  for (int i = 0; i < 10; ++i) {
    std::cout << engine() << ", ";
  }
}
```

这个例子运行结果如下：

```
1st run: 3499211612, 581869302, 3890346734, 3586334585, 545404204, 4161255391, 39229
19429, 949333985, 2715962298, 1323567403,
2nd run: 3499211612, 581869302, 3890346734, 3586334585, 545404204, 4161255391, 39229
19429, 949333985, 2715962298, 1323567403,
...
nth run: 3499211612, 581869302, 3890346734, 3586334585, 545404204, 4161255391, 39229
19429, 949333985, 2715962298, 1323567403,
```

合规代码

在该代码中，相对之前的那个有所改善，它将当前的时间作为随机数生成引擎的种子。但是，当攻击者能够控制执行设置种子的时间，该方案依然不太合适。当使用PRNG(伪随机数生成器)时，可预知的种子的值可能导致被攻击。

```
#include <ctime>
#include <random>
#include <iostream>

void f() {
  std::time_t t;
  std::mt19937 engine(std::time(&t));

  for (int i = 0; i < 10; ++i) {
    std::cout << engine() << ", ";
  }
}
```

合规代码

在此代码中，使用 `std::random_device` 生成随机值，并将其作为「Mersenne Twister」随机数生成引擎的种子。 `std::random_device` 会尽最大可能地生成不确定的随机值，但它依赖于随机数生成设备，例如 `/dev/random` 。当设备不可用时， `std::random_device` 可能会求助于随机数生成引擎；但是，生成的初始值必须足够随机才能扮演种子的角色。

```
#include <random>
#include <iostream>

void f() {
  std::random_device dev;
  std::mt19937 engine(dev());

  for (int i = 0; i < 10; ++i) {
    std::cout << engine() << ", ";
  }
}
```

这个例子运行结果如下：

```
1st run: 3921124303, 1253168518, 1183339582, 197772533, 83186419, 2599073270, 323822
2340, 101548389, 296330365, 3335314032,
2nd run: 2392369099, 2509898672, 2135685437, 3733236524, 883966369, 2529945396, 7642
22328, 138530885, 4209173263, 1693483251,
3rd run: 914243768, 2191798381, 2961426773, 3791073717, 2222867426, 1092675429, 2202
201605, 850375565, 3622398137, 422940882,
...
```

参考文献

- MSC51-CPP Ensure your random number generator is properly seeded

MSC52-CPP. 带返回值的函数必须在每个路径中返回返回值

作者：曹笑10116982 评审人：赵庆轩10146152

一个带返回值的函数必须在所有代码路径上返回返回值；否则，其行为未定义。

> *C++标准ISO/IEC 14882-2014第二章[stmt.return]明确提出：*
> *控制越过函数语句末端，其等价于返回空值；当该函数需要返回值时，其行为未定义。*

不合规代码

在该代码中，遗漏了另外一种可能的场景：当输入为正数时返回该输入的值，所以不是所有的代码路径都返回了值。

```
int absolute_value(int a) {
  if (a < 0) {
    return -a;
  }
}
```

合规代码

在该代码中，所有代码路径都返回了值。

```
int absolute_value(int a) {
```

```
  if (a < 0) {
    return -a;
  }
  return a;
}
```

不合规代码

在该代码中， `function-try-block` 处理程序并没有返回值，当异常抛出时，其行为未定义。

```
#include <vector>

std::size_t f(std::vector<int> &v, std::size_t s) try {
  v.resize(s);
  return s;
} catch (...) {
}
```

合规代码

在这个正例中， `function-try-block` 异常处理程序返回了值。

```
#include <vector>

std::size_t f(std::vector<int> &v, std::size_t s) try {
  v.resize(s);
  return s;
} catch (...) {
  return 0;
}
```

异常

MSC54-CPP-EX1: 按照C++标准ISO/IEC 14882-2014，第5章[basic.start.main]所述，当控制越过 `main` 函数语句末端，其等价于返回 `0` 值。因此，当控制越过 `main` 函数语句末端，其结果并非行为未定义。

MSC54-CPP-EX2: 当一个代码路径预期绝对不会被执行，一个标记为 `[[noreturn]]` 可以在该代码路径中被调用，或者直接抛出异常，此时允许控制路径不需要返回值，例如如

下代码所示：

```
#include <cstdlib>
#include <iostream>
[[noreturn]] void unreachable(const char *msg) {
  std::cout << "Unreachable code reached: " << msg << std::endl;
  std::exit(1);
}

enum E {
  One,
  Two,
  Three
};

int f(E e) {
  switch (e) {
  case One: return 1;
  case Two: return 2;
  case Three: return 3;
  }
  unreachable("Can never get here");
}
```

参考文献

- MSC52-CPP Value-returning functions must return a value from all exit paths

MSC54-CPP. 信号处理程序必须是POF

作者：曹笑10116982 评审人：赵庆轩10146152

C++标准ISO/IEC 14882-2014，第10章[support.runtime]，明确提出：

> *C和C++语言的共同特性的子集，包括声明，定义和表达式，它们无论在C++程序，还是C程序都是合法的。一个POF(Plain Old Function)指的就是仅使用该公共子集特性的函数，除了可能会使用普通的无锁原子操作之外，它不会直接或间接调用非POF的函数。普通的无锁原子操作指的是条款29所述的一个函数 `f` 调用；例如 `f` 并非成员函数， `f` 要么是一个 `atomic_is_lock_free` 函数，要么对于任何传递给 `f` 的原子参数 `A` ， `atomic_is_lock_free(A)` 总是为*

> *真。所有的信号处理程序应该是C连接的。在一个C++程序中，除了 `POF` 的任何函数之外，如果被信号处理程序调用，其行为依赖于实现。*

并在228页的脚注中指出：

> *特殊地，信号处理程序使用异常处理器可能存在很多问题。此外，调用 `std::exit` 引发对象的析构，包括标准库实现的对象，在信号处理程序中导致未定义的行为。*

如果一个信号处理程序不是一个POF，那么为了响应一个特定信号而调用该处理程序，其行为依赖具体实现；在最好的情况下，其结果行为未定义。所有信号处理程序必须遵循POF的定义。一方面，信号处理程序被限制在C程序中使用；另外一方面，该定义也禁止使用仅存在于C++而不是C的特性(例如非POD[Plain Old Data]对象，异常)，包括通过函数直接或间接调用该特性。

不合规代码

在代码中，信号处理函数是采用默认声明(系统要求所有的信号处理函数都应该C语言的连接的)。
但在C++代码中，函数是具有默认的C++语言连接的，当该信号处理程序被调用时，其行为未定义。

```
#include <csignal>

void sig_handler(int sig) {
  // Implementation details elided.
}

void install_signal_handler() {
  if (SIG_ERR == std::signal(SIGTERM, sig_handler)) {
    // Handle error
  }
}
```

合规代码

在此代码中， `sig_handler()` 被定义为具有C语言连接。将信号处理函数声明为C语言连接的，则信号处理程序拥有了外部连接，而不是内部连接。

```
#include <csignal>
```

```
extern "C" void sig_handler(int sig) {
  // Implementation details elided.
}

void install_signal_handler() {
  if (SIG_ERR == std::signal(SIGTERM, sig_handler)) {
    // Handle error
  }
}
```

不合规代码

在此代码中，信号处理程序调用了一个允许抛出异常的函数，并且试图处理所有抛出的异常。因为异常并非是C和C++特性的公共子集，这个例子的结果是一种依赖于具体实现的行为。
但是，实现行为不可能处理得非常恰当。例如，在一个基于栈的架构，信号是异步产生的(而不是直接调用 `std::abort` 或 `std::raise` 作为其值)，栈帧可能没有被合理地初始化，导致栈追踪不可达，并且阻止了异常被正常地捕获。

```
#include <csignal>

static void g() noexcept(false);

extern "C" void sig_handler(int sig) {
  try {
    g();
  } catch (...) {
    // Handle error
  }
}

void install_signal_handler() {
  if (SIG_ERR == std::signal(SIGTERM, sig_handler)) {
    // Handle error
  }
}
```

合规代码

目前没有一种合理的解决方案，能够让信号处理程序调用允许抛出异常的 `g()` 。即使 `g()` 的实现处理了所有异常，并且标记 `noexcept(true)` ，但是从信号处理程序调

用 `g()` 依然是不合理的，因为 `g()` 依然不是使用C和C++信号处理程序的公共子集的特性。

因此，在本代码中将 `g()` 从信号处理程序中移除，并周期性轮询一个类型为 `volatile sig_atomic_t` 的变量；如果该变量在信号处理程序中被置位为1，那么 `g()` 将会被调用，并以此响应信号。

```
#include <csignal>

volatile sig_atomic_t signal_flag = 0;
static void g() noexcept(false);

extern "C" void sig_handler(int sig) {
  signal_flag = 1;
}

void install_signal_handler() {
  if (SIG_ERR == std::signal(SIGTERM, sig_handler)) {
    // Handle error
  }
}

// Called periodically to poll the signal flag.
void poll_signal_flag() {
  if (signal_flag == 1) {
    signal_flag = 0;
    try {
      g();
    } catch(...) {
      // Handle error
    }
  }
}
```

参考文献

- MSC54-CPP A signal handler must be a plain old function