www.SourceSara.com
سایت سورس سرا - آموزش برنامه نویسی

بسمه‌تعالی

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

# عنوان مستند:

# برنامه‌نویسی امن C++

## مرکز ماهر

## تابستان ۱۳۹۶

www.SourceSara.com
سایت سورس سرا - آموزش برنامه نویسی

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران

مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

# فهرست مطالب

**فصل اول: مقدمه ..... ۱**

**۱,۱ مقدمه ..... ۲**

**۲,۱ رشد آسیب‌پذیری نرم‌افزارها ..... ۲**

**۳,۱ ضرورت برنامه‌نویسی امن ..... ۳**

**۴,۱ راهنمای سند مورد مطالعه ..... ۴**

**۵,۱ کاربرد ..... ۵**

**۶,۱ شناسه‌ها ..... ۵**

۱,۶,۱ رفتار تعریف‌نشده ..... ۵

۲,۶,۱ نمونه‌کدهای ناسازگار و راه‌حل‌های سازگار ..... ۶

۳,۶,۱ استثنائات ..... ۶

۴,۶,۱ ارزیابی خطر ..... ۶

**فصل دوم: اعلان‌ها و مقداردهی اولیه (DCL) ..... ۹**

**۱,۲ DCL50-CPP یک تابع variadic سَبک C تعریف نکنید ..... ۱۰**

۱,۱,۲ نمونه‌کد ناسازگار ..... ۱۱

۲,۱,۲ راه‌حل سازگار (توسعه‌ی بسته‌ی بازگشتی) ..... ۱۱

۳,۱,۲ راه‌حل سازگار (گسترش لیست آغازگر آماده) ..... ۱۲

۴,۱,۲ استثنائات ..... ۱۳

۵,۱,۲ ارزیابی خطر ..... ۱۴

**۲,۲ DCL51-CPP یک شناسه‌ی ذخیره‌شده را اعلان یا تعریف نکنید ..... ۱۵**

۱,۲,۲ نمونه‌کد ناسازگار (محافظ سرآیند) ..... ۱۵

۲,۲,۲ راه‌حل سازگار ..... ۱۵

۳,۲,۲ نمونه‌کد ناسازگار (عملگر تحت‌اللفظی تعریف‌شده توسط کاربر) ..... ۱۶

۴,۲,۲ راه‌حل سازگار (عملگر تحت‌اللفظی تعریف‌شده توسط کاربر) ..... ۱۶

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

www.SourceSara.com
سایت سورس سرا - آموزش برنامه نویسی



۵.۲.۲ نمونه‌کد ناسازگار (اشیای حوزه‌ی فایل) ..... ۱۶
۶.۲.۲ راه‌حل سازگار (اشیای حوزه‌ی فایل) ..... ۱۷
۷.۲.۲ نمونه‌کد ناسازگار (ماکروی ذخیره‌شده) ..... ۱۷
۸.۲.۲ راه‌حل سازگار (ماکروهای ذخیره‌شده) ..... ۱۷
۹.۲.۲ استثنائات ..... ۱۸
۱۰.۲.۲ ارزیابی خطر ..... ۱۹

**۳.۲ DCL52-CPP هرگز مرجع را از نوع const یا volatile توصیف نکنید ..... ۲۰**

۱.۳.۲ نمونه‌کد ناسازگار ..... ۲۰
۱.۱.۳.۲ جزئیات اجرا (مایکروساف ویژوال استودیو) ..... ۲۱
۲.۱.۳.۲ جزئیات اجرا (Clang) ..... ۲۱
۲.۳.۲ نمونه‌کد ناسازگار ..... ۲۱
۳.۳.۲ راه‌حل سازگار ..... ۲۱
۴.۳.۲ ارزیابی خطر ..... ۲۲

**۴.۲ DCL53-CPP اعلاناتی را که از نظر نحوی مبهم هستند، ننویسید ..... ۲۳**

۱.۴.۲ نمونه‌کد ناسازگار ..... ۲۳
۲.۴.۲ راه‌حل سازگار ..... ۲۳
۳.۴.۲ نمونه‌کد ناسازگار ..... ۲۴
۴.۴.۲ راه‌حل سازگار ..... ۲۴
۵.۴.۲ نمونه‌کد ناسازگار ..... ۲۵
۶.۴.۲ راه‌حل سازگار ..... ۲۶
۷.۴.۲ ارزیابی خطر ..... ۲۷

**5.2 DCL54-CPP توابع تخصیص و آزادسازی را به‌عنوان زوجی در یک حوزه، سربار کنید ..... ۲۸**

۱.۵.۲ نمونه‌کد ناسازگاز ..... ۲۸
۲.۵.۲ راه‌حل سازگار ..... ۲۹
۳.۵.۲ نمونه‌کد ناسازگار ..... ۳۰
۴.۵.۲ راه‌حل سازگار ..... ۳۰
۵.۵.۲ استثنائات ..... ۳۱

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکزماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

۶.۵.۲ ارزیابی خطر ..... ۳۱

**۶.۲ DCL55-CPP از نشت اطلاعات هنگام گذر از یک شی کلاس در حیطه‌ی یک مرز اعتماد، جلوگیری نمایید ..... ۳۲**

۱.۶.۲ نمونه‌کد ناسازگار ..... ۳۳
۲.۶.۲ نمونه‌کد ناسازگار ..... ۳۳
۳.۶.۲ راه‌حل سازگار ..... ۳۴
۴.۶.۲ راه‌حل سازگار (بایت‌های پدینگ) ..... ۳۵
۵.۶.۲ نمونه‌کد ناسازگار ..... ۳۵
۶.۶.۲ راه‌حل سازگار ..... ۳۷
۷.۶.۲ ارزیابی خطر ..... ۳۹
۱.۷.۶.۲ آسیب‌پذیری‌های مرتبط ..... ۳۹

**۷.۲ DCL56-CPP از ایجاد چرخه‌ها در مقداردهی اولیه‌ی اشیای ایستا بپرهیزید ..... ۴۰**

۱.۷.۲ نمونه‌کد ناسازگار ..... ۴۰
۱.۱.۷.۲ جزئیات پیاده‌سازی ..... ۴۱
۲.۷.۲ راه‌حل سازگار ..... ۴۱
۳.۷.۲ نمونه‌کد ناسازگار ..... ۴۱
۱.۳.۷.۲ جزئیات پیاده‌سازی ..... ۴۲
۴.۷.۲ راه‌حل سازگار ..... ۴۲
۵.۷.۲ ارزیابی خطر ..... ۴۳

**۸.۲ DCL57-CPP به استثنائات اجازه ندهید از توابع مخرب یا آزادساز فرار کنند ..... ۴۴**

۱.۸.۲ نمونه‌کد ناسازگار ..... ۴۴
۲.۸.۲ نمونه‌کد ناسازگار (*std::uncaught_exception ()*) ..... ۴۵
۳.۸.۲ نمونه‌کد ناسازگار (*function_try_block*) ..... ۴۵
۴.۸.۲ راه‌حل سازگار ..... ۴۶
۵.۸.۲ نمونه‌کد ناسازگار ..... ۴۷
۶.۸.۲ راه‌حل سازگار ..... ۴۷
۷.۸.۲ ارزیابی خطر ..... ۴۸

مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

www.SourceSara.com
سایت سورس سرا - آموزش برنامه نویسی



**۹,۲ DCL58-CPP فضاهای استاندارد را تغییر ندهید ..... ۴۹**

۱,۹,۲ نمونه‌کد ناسازگار ..... ۴۹

۲,۹,۲ راه‌حل سازگار ..... ۵۰

۳,۹,۲ نمونه‌کد ناسازگار ..... ۵۰

۴,۹,۲ راه‌حل سازگار ..... ۵۱

۵,۹,۲ راه‌حل سازگار ..... ۵۱

۶,۹,۲ ارزیابی خطر ..... ۵۲

**۱۰,۲ DCL59-CPP یک فضای بدون نام را در یک فایل سرآیند تعریف نکنید ..... ۵۳**

۱,۱۰,۲ نمونه‌کد ناسازگار ..... ۵۳

۲,۱۰,۲ راه‌حل سازگار ..... ۵۴

۳,۱۰,۲ نمونه‌کد ناسازگار ..... ۵۵

۴,۱۰,۲ راه‌حل سازگار ..... ۵۶

۵,۱۰,۲ نمونه‌کد ناسازگار ..... ۵۷

۶,۱۰,۲ نمونه‌کد ناسازگار ..... ۵۷

۷,۱۰,۲ راه‌حل سازگار ..... ۵۸

۸,۱۰,۲ ارزیابی خطر ..... ۵۸

**۱۱,۲ DCL60-CPP از قانوت تک‌توصیفی پیروی کنید ..... ۵۹**

۱,۱۱,۲ نمونه‌کد ناسازگار ..... ۵۹

۲,۱۱,۲ راه‌حل سازگار ..... ۶۰

۳,۱۱,۲ راه‌حل سازگار ..... ۶۰

۴,۱۱,۲ نمونه‌کد ناسازگار (مایکروسافت ویژوال استودیو) ..... ۶۱

۱,۴,۱۱,۲ جزئیات پیاده‌سازی ..... ۶۱

۵,۱۱,۲ راه‌حل سازگار ..... ۶۲

۶,۱۱,۲ نمونه‌کد ناسازگار ..... ۶۲

۷,۱۱,۲ راه‌حل سازگار ..... ۶۳

۸,۱۱,۲ راه‌حل سازگار ..... ۶۳

۹,۱۱,۲ ارزیابی خطر ..... ۶۴

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

**فصل سوم: اصطلاحات (EXP) ..... ۶۵**

**۱.۳ EXP50-CPP به ترتیب ارزیابی برای تأثیرات جانبی وابسته نباشید ..... ۶۶**

۱.۱.۳ نمونه‌کد ناسازگار ..... ۶۷

۲.۱.۳ راه‌حل سازگار ..... ۶۸

۳.۱.۳ نمونه‌کد ناسازگار ..... ۶۸

۴.۱.۳ راه‌حل سازگار ..... ۶۹

۵.۱.۳ نمونه‌کد ناسازگار ..... ۶۹

۶.۱.۳ راه‌حل سازگار ..... ۶۹

۷.۱.۳ نمونه‌کد ناسازگار ..... ۷۰

۸.۱.۳ راه‌حل سازگار ..... ۷۰

۹.۱.۳ ارزیابی خطر ..... ۷۱

**۲.۳ EXP51-CPP یک آرایه را توسط یک اشاره‌گر نادرست حذف نکنید ..... ۷۲**

۱.۲.۳ نمونه‌کد ناسازگار ..... ۷۲

۲.۲.۳ راه‌حل سازگار ..... ۷۲

۳.۲.۳ ارزیابی خطر ..... ۷۳

**۳.۳ EXP52-CPP در عملوندهای ارزیابی نشده، به تاثیرات جانبی وابسته نباشید ..... ۷۴**

۱.۳.۳ نمونه‌کد ناسازگار ..... ۷۴

۲.۳.۳ راه‌حل سازگار (*sizeof*) ..... ۷۴

۳.۳.۳ نمونه‌کد ناسازگار (*decltype*) ..... ۷۵

۴.۳.۳ راه‌حل سازگار (*decltype*) ..... ۷۵

۵.۳.۳ استثنائات ..... ۷۵

۶.۳.۳ ارزیابی خطر ..... ۷۷

**۴.۳ EXP53-CPP حافظه‌ی مقداردهی اولیه‌نشده را نخوانید ..... ۷۸**

۱.۴.۳ نمونه‌کد ناسازگار ..... ۷۹

۲.۴.۳ راه‌حل سازگار ..... ۷۹

۳.۴.۳ نمونه‌کد ناسازگار ..... ۷۹

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

۴.۴.۳ راه‌حل سازگار ..... ۷۹
۵.۴.۳ نمونه‌کد ناسازگار ..... ۸۰
۶.۴.۳ راه‌حل سازگار ..... ۸۰
۷.۴.۳ ارزیابی خطر ..... ۸۱
**۵.۳ EXP54-CPP پس از طول عمر یک شی، به آن دسترسی پیدا نکنید ..... ۸۲**
۱.۵.۳ نمونه‌کد ناسازگار ..... ۸۲
۲.۵.۳ راه‌حل سازگار ..... ۸۲
۳.۵.۳ نمونه‌کد ناسازگار ..... ۸۳
۴.۵.۳ راه‌حل سازگار ..... ۸۳
۵.۵.۳ نمونه‌کد ناسازگار ..... ۸۳
۶.۵.۳ راه‌حل سازگار ..... ۸۴
۷.۵.۳ نمونه‌کد ناسازگار ..... ۸۴
۸.۵.۳ راه‌حل سازگار ..... ۸۵
۹.۵.۳ نمونه‌کد ناسازگار ..... ۸۶
۱۰.۵.۳ راه‌حل سازگار ..... ۸۶
۱۱.۵.۳ نمونه‌کد ناسازگار ..... ۸۷
۱۲.۵.۳ راه‌حل سازگار ..... ۸۷
۱۳.۵.۳ نمونه‌کد ناسازگار ..... ۸۸
۱,۱۳,۵,۳ جزییات پیاده‌سازی ..... ۸۹
۱۴.۵.۳ راه‌حل سازگار (*std::uninitialized_copy()*) ..... ۸۹
۱۵.۵.۳ راه‌حل سازگار (*std::raw_storage_iterator*) ..... ۸۹
۱۶.۵.۳ ارزیابی خطر ..... ۸۹
**۶.۳ EXP55-CPP به یک شی cv-qualified توسط یک شی cv-unqualified دسترسی پیدا نکنید ..... ۹۱**
۱.۶.۳ نمونه‌کد ناسازگار ..... ۹۱
۲.۶.۳ راه‌حل سازگار ..... ۹۱
۳.۶.۳ نمونه‌کد ناسازگار ..... ۹۲
۴.۶.۳ راه‌حل سازگار ..... ۹۲

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

۵.۶.۳ نمونه‌کد ناسازگار ..... ۹۳
۶.۶.۳ راه‌حل سازگار ..... ۹۳
۷.۶.۳ استثنائات ..... ۹۴
۸.۶.۳ ارزیابی خطر ..... ۹۴

**۷.۳ EXP56-CPP یک تابع را توسط یک پیوند زبانی نامنطبق فراخوانی نکنید ..... ۹۵**

۱.۷.۳ نمونه‌کد ناسازگار ..... ۹۵
۲.۷.۳ راه‌حل سازگار ..... ۹۶
۳.۷.۳ ارزیابی خطر ..... ۹۶

**۸.۳ EXP57-CPP اشاره‌گرهای کلاسهای ناقص را تبدیل و یا حذف نکنید ..... ۹۷**

۱.۸.۳ نمونه‌کد ناسازگار ..... ۹۷
۲.۸.۳ راه‌حل سازگار (حذف) ..... ۹۷
۳.۸.۳ راه‌حل سازگار ..... ۹۸
۴.۸.۳ نمونه‌کد ناسازگار ..... ۹۸
۱.۴.۸.۳ جزئیات پیاده‌سازی ..... ۱۰۰
۵.۸.۳ راه‌حل سازگار ..... ۱۰۰
۶.۸.۳ ارزیابی خطر ..... ۱۰۱

**۹.۳ EXP58-CPP از یک شی نوع درست به *va_start* گذر کنید ..... ۱۰۲**

۱.۹.۳ نمونه‌کد ناسازگار ..... ۱۰۲
۲.۹.۳ راه‌حل سازگار ..... ۱۰۲
۳.۹.۳ نمونه‌کد ناسازگار ..... ۱۰۳
۴.۹.۳ راه‌حل سازگار ..... ۱۰۳
۵.۹.۳ نمونه‌کد ناسازگار ..... ۱۰۴
۶.۹.۳ راه‌حل سازگار ..... ۱۰۴
۷.۹.۳ ارزیابی خطر ..... ۱۰۴

**۱۰.۳ EXP59-CPP از *offsetof()* در ارتباط با انواع و اعضای معتبر استفاده کنید ..... ۱۰۵**

۱.۱۰.۳ نمونه‌کد ناسازگار ..... ۱۰۵
۱.۱.۱۰.۳ جزئیات پیاده‌سازی ..... ۱۰۵

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

۲.۱۰.۳ راه‌حل سازگار ..... ۱۰۵
۳.۱۰.۳ نمونه‌کد ناسازگار ..... ۱۰۶
۱.۳.۱۰.۳ جزئیات پیاده‌سازی ..... ۱۰۶
۴.۱۰.۳ راه‌حل سازگار ..... ۱۰۷
۵.۱۰.۳ ارزیابی خطر ..... ۱۰۷
**۱۱.۳ EXP60-CPP شی نوع لایه‌ای غیراستاندارد را از مرزهای اجرا عبور ندهید ..... ۱۰۸**
۱.۱۱.۳ نمونه‌کد ناسازگار ..... ۱۰۹
۲.۱۱.۳ راه‌حل سازگار ..... ۱۱۰
۳.۱۱.۳ نمونه‌کد ناسازگار ..... ۱۱۰
۴.۱۱.۳ راه‌حل سازگار ..... ۱۱۱
۵.۱۱.۳ ارزیابی خطر ..... ۱۱۱
**۱۲.۳ EXP61-CPP نباید یک شی لاندا بیشتر از اشیایی رجوع‌شده به خود، عمر کند ..... ۱۱۳**
۱.۱۲.۳ نمونه‌کد ناسازگار ..... ۱۱۳
۲.۱۲.۳ راه‌حل سازگار ..... ۱۱۴
۳.۱۲.۳ نمونه‌کد ناسازگار ..... ۱۱۴
۴.۱۲.۳ راه‌حل سازگار ..... ۱۱۵
۵.۱۲.۳ ارزیابی خطر ..... ۱۱۵
**۱۳.۳ EXP62-CPP به بیت‌های یک ارائه‌ی شی، که بخشی از ارائه‌ی مقدار شی نیست، دسترسی پیدا نکنید ..... ۱۱۷**
۱.۱۳.۳ نمونه‌کد ناسازگار ..... ۱۱۷
۲.۱۳.۳ راه‌حل سازگار ..... ۱۱۸
۳.۱۳.۳ نمونه‌کد ناسازگار ..... ۱۱۹
۴.۱۳.۳ راه‌حل سازگار ..... ۱۱۹
۵.۱۳.۳ استثنائات ..... ۱۱۹
۶.۱۳.۳ ارزیابی خطر ..... ۱۲۰
**۱۴.۳ EXP63-CPP به مقدار یک شی انتقال‌یافته تکیه نکنید ..... ۱۲۱**

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

۱,۱۴,۳ نمونه‌کد ناسازگار ..... ۱۲۱
۱,۱,۱۴,۳ جزئیات پیاده‌سازی ..... ۱۲۲
۲,۱۴,۳ راه‌حل سازگار ..... ۱۲۳
۳,۱۴,۳ نمونه‌کد ناسازگار ..... ۱۲۳
۴,۱۴,۳ راه‌حل سازگار ..... ۱۲۳
۵,۱۴,۳ راه‌حل سازگار ..... ۱۲۴
۶,۱۴,۳ ارزیابی خطر ..... ۱۲۴

**فصل چهارم: اعداد صحیح (INT) ..... ۱۲۵**

**۱,۴ INT50-CPP به مقدار شمارشی خارج از محدوده تبدیل نکنید ..... ۱۲۶**
۱,۱,۴ نمونه‌کد ناسازگار (بررسی کران‌ها) ..... ۱۲۶
۲,۱,۴ راه‌حل سازگار (بررسی کران‌ها) ..... ۱۲۷
۳,۱,۴ راه‌حل سازگار (مقدار شمارشی دارای قلمرو) ..... ۱۲۷
۴,۱,۴ راه‌حل سازگار (مقدار شمارشی بدون قلمرو ثابت) ..... ۱۲۸
۵,۱,۴ ارزیابی خطر ..... ۱۲۸

**فصل پنجم: کاراکترها و رشته‌ها (STR) ..... ۱۲۹**

**۱,۵ STR50-CPP تضمین کنید که ذخیره‌سازی برای رشته‌ها، دارای فضای کافی برای داده‌های کاراکتری و کاراکتر تهی است ..... ۱۳۰**
۱,۱,۵ نمونه‌کد ناسازگار ..... ۱۳۰
۲,۱,۵ نمونه‌کد ناسازگار ..... ۱۳۰
۳,۱,۵ راه‌حل سازگار ..... ۱۳۱
۴,۱,۵ نمونه‌کد ناسازگار ..... ۱۳۱
۵,۱,۵ راه‌حل سازگار ..... ۱۳۲
۶,۱,۵ ارزیابی خطر ..... ۱۳۳

**۲,۵ STR51-CPP اقدام به ساخت یک *std::string* از یک اشاره‌گر تهی نکنید ..... ۱۳۴**
۱,۱,۲,۵ جزئیات پیاده‌سازی ..... ۱۳۵
۲,۲,۵ نمونه‌کد ناسازگار ..... ۱۳۵

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکزماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

۳.۲.۵ راه‌حل سازگار ..... ۱۳۶
۴.۲.۵ ارزیابی خطر ..... ۱۳۶

**۳.۵ STR52-CPP از ارجاعات، اشاره‌گرها و تکرارکننده‌های معتبر برای ارجاع به عناصر یک *basic_string* استفاده کنید ..... ۱۳۷**

۱.۳.۵ نمونه‌کد ناسازگار ..... ۱۳۷
۲.۳.۵ راه‌حل سازگار (*std::string::insert()*) ..... ۱۳۸
۳.۳.۵ راه‌حل سازگار (*std::replace()*) ..... ۱۳۸
۴.۳.۵ نمونه‌کد ناسازگار ..... ۱۳۸
۵.۳.۵ راه‌حل سازگار ..... ۱۳۹
۶.۳.۵ ارزیابی خطر ..... ۱۳۹

**۴.۵ STR53-CPP دسترسی به مؤلفه‌ی بررسی را محدوده‌بندی کنید ..... ۱۴۰**

۱.۴.۵ نمونه‌کد ناسازگار ..... ۱۴۰
۲.۴.۵ راه‌حل سازگار (*try/catch*) ..... ۱۴۰
۳.۴.۵ راه‌حل سازگار (بررسی بازه) ..... ۱۴۱
۴.۴.۵ نمونه‌کد ناسازگار ..... ۱۴۱
۵.۴.۵ راه‌حل سازگار ..... ۱۴۲
۶.۴.۵ ارزیابی خطر ..... ۱۴۲

**فصل ششم: مدیریت حافظه (MEM) ..... ۱۴۳**

**۱.۶ MEM50-CPP به حافظه‌ی آزاد دسترسی پیدا نکنید ..... ۱۴۴**

۱.۱.۶ نمونه‌کد ناسازگار (*new* و *delete*) ..... ۱۴۴
۲.۱.۶ راه‌حل سازگار (*new* و *delete*) ..... ۱۴۵
۳.۱.۶ راه‌حل سازگار (دوره‌ی ذخیره‌سازی خودکار) ..... ۱۴۵
۴.۱.۶ نمونه‌کد ناسازگار (*std::unique_ptr*) ..... ۱۴۵
۵.۱.۶ راه‌حل سازگار (*std::unique_ptr*) ..... ۱۴۶
۶.۱.۶ راه‌حل سازگار ..... ۱۴۷
۷.۱.۶ نمونه‌کد ناسازگار (*std::string::c_str()*) ..... ۱۴۷

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه‌ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکزماهر
مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

۸.۱.۶ راه‌حل سازگار (*std::string::c_str()*) ..... ۱۴۸
۹.۱.۶ نمونه‌کد ناسازگار ..... ۱۴۸
۱۰.۱.۶ راه‌حل سازگار ..... ۱۴۸
۱۱.۱.۶ راه‌حل سازگار ..... ۱۴۹
۱۲.۱.۶ ارزیابی خطر ..... ۱۴۹

**۲.۶ MEM51-CPP منابع تخصیص‌یافته را درست آزاد کنید ..... ۱۵۰**

۱.۲.۶ نمونه‌کد ناسازگار (جایابی *new*) ..... ۱۵۰
۲.۲.۶ راه‌حل سازگار (جایابی *new*) ..... ۱۵۱
۳.۲.۶ نمونه‌کد ناسازگار (*delete* مقداردهی اولیه‌نشده) ..... ۱۵۲
۴.۲.۶ راه‌حل سازگار (*delete* مقداردهی اولیه‌نشده) ..... ۱۵۲
۵.۲.۶ نمونه‌کد ناسازگار (آرایه‌ی *new[]*) ..... ۱۵۳
۶.۲.۶ راه‌حل سازگار (آرایه‌ی *new[]*) ..... ۱۵۳
۷.۲.۶ نمونه‌کد ناسازگار (*malloc ()*) ..... ۱۵۳
۱.۷.۲.۶ جزئیات پیاده‌سازی ..... ۱۵۳
۸.۲.۶ راه‌حل سازگار (*malloc ()*) ..... ۱۵۴
۹.۲.۶ نمونه‌کد ناسازگار (*new*) ..... ۱۵۴
۱۰.۲.۶ راه‌حل سازگار (*new*) ..... ۱۵۴
۱۱.۲.۶ نمونه‌کد ناسازگار (کلاس *new*) ..... ۱۵۵
۱۲.۲.۶ راه‌حل سازگار (کلاس *new*) ..... ۱۵۵
۱۳.۲.۶ نمونه‌کد ناسازگار (*std::unique_ptr*) ..... ۱۵۶
۱۴.۲.۶ راه‌حل سازگار (*std::unique_ptr*) ..... ۱۵۶
۱۵.۲.۶ نمونه‌کد ناسازگار (*std::shared_ptr*) ..... ۱۵۶
۱۶.۲.۶ راه‌حل سازگار (*std::shared_ptr*) ..... ۱۵۷
۱۷.۲.۶ ارزیابی خطر ..... ۱۵۷

**۳.۶ MEM52-CPP خطاهای تخصیص حافظه را تشخیص دهید و به آنها رسیدگی کنید ..... ۱۵۸**

۱.۳.۶ نمونه‌کد ناسازگار ..... ۱۵۸
۲.۳.۶ راه‌حل سازگار (*std::nothrow*) ..... ۱۵۹

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

۳.۳.۶ راه‌حل سازگار (*std::bad_alloc*) ..... ۱۵۹
۴.۳.۶ راه‌حل سازگار (*noexcept(false)*) ..... ۱۶۰
۵.۳.۶ نمونه‌کد ناسازگار ..... ۱۶۰
۶.۳.۶ راه‌حل سازگار (*std::unique_ptr*) ..... ۱۶۱
۷.۳.۶ راه‌حل سازگار (مراجع) ..... ۱۶۱
۸.۳.۶ ارزیابی خطر ..... ۱۶۱

**۴.۶ MEM53-CPP وقتی طول عمر اشیا به‌صورت دستی مدیریت می‌شود، به‌صراحت اشیا را بسازید و از بین ببرید ..... ۱۶۳**

۱.۴.۶ نمونه‌کد ناسازگار ..... ۱۶۴
۲.۴.۶ راه‌حل سازگار ..... ۱۶۴
۳.۴.۶ نمونه‌کد ناسازگار ..... ۱۶۵
۴.۴.۶ راه‌حل سازگار ..... ۱۶۶
۵.۴.۶ استثنائات ..... ۱۶۷
۶.۴.۶ ارزیابی خطر ..... ۱۶۷

**۵.۶ MEM54-CPP توسط اشاره‌گر مرتب مناسب، جایابی جدید را برای ظرفیت ذخیره‌سازی کافی فراهم کنید ..... ۱۶۸**

۱.۵.۶ نمونه‌کد ناسازگار ..... ۱۶۸
۲.۵.۶ نمونه‌کد ناسازگار ..... ۱۶۹
۳.۵.۶ راه‌حل سازگار (*alignas*) ..... ۱۶۹
۴.۵.۶ راه‌حل سازگار (*std::aligned_storage*) ..... ۱۶۹
۵.۵.۶ نمونه‌کد ناسازگار (شکست در محاسبه‌ی سربار آرایه) ..... ۱۷۰
۶.۵.۶ راه‌حل سازگار (Clang/GCC) ..... ۱۷۰
۷.۵.۶ ارزیابی خطر ..... ۱۷۱

**۶.۶ MEM55-CPP به نیازمندی‌های مدیریت ذخیره‌سازی پویای جایگزینی اولویت دهید ..... ۱۷۲**

۱.۶.۶ نمونه‌کد ناسازگار ..... ۱۷۲
۲.۶.۶ راه‌حل سازگار ..... ۱۷۲
۳.۶.۶ ارزیابی خطر ..... ۱۷۳

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

**۷.۶ MEM56-CPP یک مقدار اشاره‌گر از پیش مالکیت‌یافته را در یک اشاره‌گر هوشمند غیرمرتبط ذخیره نکنید ..... ۱۷۴**

۱.۷.۶ نمونه‌کد ناسازگار ..... ۱۷۴

۲.۷.۶ راه‌حل سازگار ..... ۱۷۴

۳.۷.۶ ارزیابی خطر ..... ۱۷۵

**فصل هفتم: ورودی و خروجی (FIO) ..... ۱۷۶**

**۱.۷ FIO50-CPP به‌طور متناوب و بدون مداخله‌ی یک فراخوانی مکان‌یابی، از یک جریان فایل، ورودی و خروجی نگیرید ..... ۱۷۷**

۱.۱.۷ نمونه‌کد ناسازگار ..... ۱۷۸

۲.۱.۷ راه‌حل سازگار ..... ۱۷۸

۳.۱.۷ ارزیابی خطر ..... ۱۷۸

**۲.۷ FIO51-CPP وقتی دیگر به فایل‌ها نیاز ندارید، آنها را ببندید ..... ۱۸۰**

۱.۲.۷ نمونه‌کد ناسازگار ..... ۱۸۰

۲.۲.۷ راه‌حل سازگار ..... ۱۸۱

۳.۲.۷ راه‌حل سازگار ..... ۱۸۱

۴.۲.۷ ارزیابی خطر ..... ۱۸۲

**فصل هشتم: اداره کردن استثنائات و خطاها (ERR) ..... ۱۸۳**

**۱.۸ ERR50-CPP به یک‌باره برنامه را خاتمه ندهید ..... ۱۸۴**

۱.۱.۸ نمونه‌کد ناسازگار ..... ۱۸۶

۲.۱.۸ راه‌حل سازگار ..... ۱۸۶

۳.۱.۸ استثنائات ..... ۱۸۷

۴.۱.۸ ارزیابی خطر ..... ۱۸۷

**۲.۸ ERR51-CPP تمامی استثنائات را اداره کنید ..... ۱۸۹**

۱.۲.۸ نمونه‌کد ناسازگار ..... ۱۸۹

۲.۲.۸ راه‌حل سازگار ..... ۱۸۹

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

۳,۲,۸ نمونه‌کد ناسازگار ..... ۱۹۰
۴,۲,۸ راه‌حل سازگار ..... ۱۹۰
۵,۲,۸ ارزیابی خطر ..... ۱۹۱

**۳,۸ ERR52-CPP از *setjmp()* یا *longjmp()* استفاده نکنید ..... ۱۹۲**
۱,۳,۸ نمونه‌کد ناسازگار ..... ۱۹۲
۱,۱,۳,۸ جزئیات پیاده‌سازی ..... ۱۹۳
۲,۳,۸ راه‌حل سازگار ..... ۱۹۳
۱,۲,۳,۸ جزئیات پیاده‌سازی ..... ۱۹۴
۳,۳,۸ ارزیابی خطر ..... ۱۹۴

**۴,۸ ERR53-CPP به کلاس‌های پایه و یا اعضای داده‌ای کلاس را در یک اداره‌کننده‌ی بلاک *try* سازنده یا مخرب ارجاع ندهید ..... ۱۹۵**
۱,۴,۸ نمونه‌کد ناسازگار ..... ۱۹۵
۲,۴,۸ راه‌حل سازگار ..... ۱۹۵
۳,۴,۸ ارزیابی خطر ..... ۱۹۶

**۵,۸ ERR54-CPP اداره‌کنندگان *catch* باید انواع پارامتری خود را از بیشترین اشتقاقی تا کمترین اشتقاقی، مرتب کنند ..... ۱۹۷**
۱,۵,۸ نمونه‌کد ناسازگار ..... ۱۹۷
۲,۵,۸ راه‌حل سازگار ..... ۱۹۷
۳,۵,۸ ارزیابی خطر ..... ۱۹۸

**۶,۸ ERR55-CPP به مشخصات استثنا اولویت دهید ..... ۱۹۹**
۱,۶,۸ نمونه‌کد ناسازگار ..... ۱۹۹
۲,۶,۸ راه‌حل سازگار ..... ۲۰۰
۳,۶,۸ راه‌حل سازگار ..... ۲۰۰
۴,۶,۸ نمونه‌کد ناسازگار ..... ۲۰۱
۵,۶,۸ راه‌حل سازگار ..... ۲۰۱
۱,۵,۶,۸ جزئیات پیاده‌سازی ..... ۲۰۱

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

۶.۶.۸ ارزیابی خطر ..... ۲۰۲

**۷.۸ ERR56-CPP ایمنی استثنا را تضمین کنید ..... ۲۰۳**

۱.۷.۸ نمونه‌کد ناسازگار (بدون ایمنی از استثنا) ..... ۲۰۴

۲.۷.۸ راه‌حل سازگار (ایمنی استثنای قوی) ..... ۲۰۵

۳.۷.۸ ارزیابی خطر ..... ۲۰۶

**۸.۸ ERR57-CPP هنگام اداره‌ی استثنائات، منابع را نشت ندهید ..... ۲۰۸**

۱.۸.۸ نمونه‌کد ناسازگار ..... ۲۰۸

۲.۸.۸ راه‌حل سازگار (*delete*) ..... ۲۰۹

۳.۸.۸ راه‌حل سازگار (الگوی طراحی RAII) ..... ۲۱۰

۴.۸.۸ نمونه‌کد ناسازگار ..... ۲۱۱

۵.۸.۸ راه‌حل سازگار (*try/catch*) ..... ۲۱۲

۶.۸.۸ راه‌حل سازگار (*std::unique_ptr*) ..... ۲۱۲

۷.۸.۸ ارزیابی خطر ..... ۲۱۳

**۹.۸ ERR58-CPP پیش از آغاز اجرای *main()*، تمامی استثنائات را اداره نمایید ..... ۲۱۴**

۱.۹.۸ نمونه‌کد ناسازگار ..... ۲۱۴

۲.۹.۸ راه‌حل سازگار ..... ۲۱۴

۳.۹.۸ نمونه‌کد ناسازگار ..... ۲۱۵

۴.۹.۸ راه‌حل سازگار ..... ۲۱۶

۵.۹.۸ ارزیابی خطر ..... ۲۱۶

**۱۰.۸ ERR59-CPP استثنایی را خارج از مرزهای اجرا ایجاد نکنید ..... ۲۱۷**

۱.۱۰.۸ نمونه‌کد ناسازگار ..... ۲۱۷

۲.۱۰.۸ راه‌حل سازگار ..... ۲۱۸

۳.۱۰.۸ ارزیابی خطر ..... ۲۱۹

**۱۱.۸ ERR60-CPP اشیای استثنا نباید از طریق کپی قابل ساخت، ایجاد شوند ..... ۲۲۰**

۱.۱۱.۸ نمونه‌کد ناسازگار ..... ۲۲۰

۲.۱۱.۸ راه‌حل سازگار ..... ۲۲۱

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

۳.۱۱.۸ راه‌حل سازگار ..... ۲۲۲
۴.۱۱.۸ ارزیابی خطر ..... ۲۲۳

**۱۲.۸ ERR61-CPP استثناها را به‌وسیله‌ی ارجاع مقدار سمت چپ بگیرید ..... ۲۲۴**

۱.۱۲.۸ نمونه‌کد ناسازگار ..... ۲۲۴
۲.۱۲.۸ راه‌حل سازگار ..... ۲۲۵
۳.۱۲.۸ ارزیابی خطر ..... ۲۲۶

**۱۳.۸ ERR62-CPP خطاها را هنگام تبدیل یک رشته به یک عدد تشتخیص دهید ..... ۲۲۷**

۱.۱۳.۸ نمونه‌کد ناسازگار ..... ۲۲۷
۲.۱۳.۸ راه‌حل سازگار ..... ۲۲۸
۳.۱۳.۸ راه‌حل سازگار ..... ۲۲۸
۴.۱۳.۸ ارزیابی خطر ..... ۲۲۹

**فصل نهم: برنامه‌نویسی شی‌گرا (OOP) ..... ۲۳۰**

**۱.۹ OOP50-CPP توابع مجازی را از طریق سازنده‌ها یا مخرب‌ها فراخوانی نکنید ..... ۲۳۱**

۱.۱.۹ نمونه‌کد ناسازگار ..... ۲۳۱
۲.۱.۹ راه‌حل سازگار ..... ۲۳۲
۳.۱.۹ استثنائات ..... ۲۳۴
۴.۱.۹ ارزیابی خطر ..... ۲۳۵

**۲.۹ OOP51-CPP اشیای اشتقاقی را نبُرید ..... ۲۳۶**

۱.۲.۹ نمونه‌کد ناسازگار ..... ۲۳۶
۲.۲.۹ راه‌حل سازگار (اشاره‌گرها) ..... ۲۳۷
۳.۲.۹ راه‌حل سازگار (مراجع) ..... ۲۳۸
۴.۲.۹ راه‌حل سازگار (غیر قابل نسخه‌برداری) ..... ۲۳۹
۵.۲.۹ نمونه‌کد ناسازگار ..... ۲۴۱
۶.۲.۹ راه‌حل سازگار ..... ۲۴۱
۷.۲.۹ ارزیابی خطر ..... ۲۴۲

**۳.۹ OOP52-CPP یک شی چندریختی را بدون مخرب مجازی حذف نکنید ..... ۲۴۳**

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

www.SourceSara.com
سایت سورس سرا - آموزش برنامه نویسی



۱.۳.۹ نمونه‌کد ناسازگار ..... ۲۴۳
۲.۳.۹ نمونه‌کد ناسازگار ..... ۲۴۳
۳.۳.۹ راه‌حل سازگار ..... ۲۴۴
۴.۳.۹ ارزیابی خطر ..... ۲۴۴

**۴.۹ OOP53-CPP مقداردهنده‌های اولیه‌ی اعضای سازنده را به‌ترتیب استاندارد بنویسید ..... ۲۴۵**

۱.۴.۹ نمونه‌کد ناسازگار ..... ۲۴۵
۲.۴.۹ راه‌حل سازگار ..... ۲۴۶
۳.۴.۹ نمونه‌کد ناسازگار ..... ۲۴۶
۴.۴.۹ راه‌حل سازگار ..... ۲۴۷
۵.۴.۹ ارزیابی خطر ..... ۲۴۷

**۵.۹ OOP54-CPP به‌خوبی به تخصیص خودکپی رسیدگی کنید ..... ۲۴۸**

۱.۵.۹ نمونه‌کد ناسازگار ..... ۲۴۹
۲.۵.۹ راه‌حل سازگار (خودآزمایی) ..... ۲۵۰
۳.۵.۹ راه‌حل سازگار (کپی و تعویض) ..... ۲۵۱
۴.۵.۹ ارزیابی خطر ..... ۲۵۲

**6.9 OOP55-CPP برای دسترسی به اعضایی که وجود ندارند، از عملگرهای اشاره‌گر به عضو استفاده نکنید ..... ۲۵۳**

۱.۶.۹ نمونه‌کد ناسازگار ..... ۲۵۴
۲.۶.۹ راه‌حل سازگار ..... ۲۵۵
۳.۶.۹ نمونه‌کد ناسازگار ..... ۲۵۵
۴.۶.۹ راه‌حل سازگار ..... ۲۵۶
۵.۶.۹ ارزیابی خطر ..... ۲۵۷

**۷.۹ OOP56-CPP به نیازمندی‌های نگهدارنده‌ی جایگزینی احترام بگذارید ..... ۲۵۸**

۱.۷.۹ نمونه‌کد ناسازگار ..... ۲۵۸
۲.۷.۹ راه‌حل سازگار ..... ۲۵۹
۳.۷.۹ ارزیابی خطر ..... ۲۶۰

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

**۸.۹ OOP57-CPP توابع عضو خاص و عملگرهای دچار سربارشده را به توابع کتابخانه‌ی استاندارد C ترجیح دهید ..... ۲۶۱**

۱.۸.۹ نمونه‌کد ناسازگار ..... ۲۶۱
۲.۸.۹ راه‌حل سازگار ..... ۲۶۲
۳.۸.۹ نمونه‌کد ناسازگار ..... ۲۶۳
۴.۸.۹ راه‌حل سازگار ..... ۲۶۴
۵.۸.۹ نمونه‌کد ناسازگار ..... ۲۶۵
۶.۸.۹ راه‌حل سازگار ..... ۲۶۶
۷.۸.۹ ارزیابی خطر ..... ۲۶۷

**۹.۹ OOP58-CPP عملیات‌های کپی نباید شی منبع را جهش دهند ..... ۲۶۸**

۱.۹.۹ نمونه‌کد ناسازگار ..... ۲۶۸
۲.۹.۹ راه‌حل سازگار ..... ۲۶۹
۳.۹.۹ ارزیابی خطر ..... ۲۷۰

**فصل دهم: همزمانی (CON) ..... ۲۷۱**

**۱.۱۰ CON50-CPP یک میوتکس قفل‌شده را از بین نبرید ..... ۲۷۲**

۱.۱.۱۰ نمونه‌کد ناسازگار ..... ۲۷۲
۲.۱.۱۰ راه‌حل سازگار ..... ۲۷۳
۳.۱.۱۰ راه‌حل سازگار ..... ۲۷۳
۴.۱.۱۰ ارزیابی خطر ..... ۲۷۴

**۲.۱۰ CON51-CPP مطمئن شوید که قفل‌های برگزارشده‌ی فعال، در شرایط استثنایی آزاد می‌شوند. ۲۷۵**

۱.۲.۱۰ نمونه‌کد ناسازگار ..... ۲۷۵
۲.۲.۱۰ راه‌حل سازگار (آزادسازی دستی) ..... ۲۷۶
۳.۲.۱۰ راه‌حل سازگار (شی قفل) ..... ۲۷۶
۴.۲.۱۰ ارزیابی خطر ..... ۲۷۷

**۳.۱۰ CON52-CPP به هنگام دسترسی به فیلدهای بیتی چندین نخ، از رقابت‌های داده‌ای بپرهیزید ... ۲۷۸**

۱.۳.۱۰ نمونه‌کد ناسازگار ..... ۲۷۸

مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

مرکز ماهر
مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای
وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران

۲.۳.۱۰ راه‌حل سازگار (فیلد بیتی C++ 11 و سپس، میوتکس) ..... ۲۷۹
۳.۳.۱۰ راه‌حل سازگار( C++ 11 ) ..... ۲۸۰
۴.۳.۱۰ ارزیابی خطر ..... ۲۸۱

**۴.۱۰ CON53-CPP با قفل کردن به ترتیب از پیش تعیین‌شده، از بن بست اجتناب کنید ..... ۲۸۲**

۱.۴.۱۰ نمونه‌کد ناسازگار ..... ۲۸۲
۲.۴.۱۰ راه‌حل سازگار (مرتب‌سازی دستی) ..... ۲۸۳
۳.۴.۱۰ راه‌حل سازگار (*std::lock()*) ..... ۲۸۵
۴.۴.۱۰ ارزیابی خطر ..... ۲۸۶

**۵.۱۰ CON54-CPP توابعی را که ممکن است به‌صورت جعلی در یک حلقه پدیدار شوند، بپوشانید ... ۲۸۷**

۱.۵.۱۰ نمونه‌کد ناسازگار ..... ۲۸۸
۲.۵.۱۰ راه‌حل سازگار ..... ۲۸۹
۳.۵.۱۰ راه‌حل سازگار (حلقه‌ی صریح با گزاره‌ی لاندا) ..... ۲۸۹
۴.۵.۱۰ ارزیابی خطر ..... ۲۹۰

**۶.۱۰ CON55-CPP هنگام استفاده از متغیرهای شرطی، ایمنی و حیات نخ را حفظ کنید ..... ۲۹۱**

۱.۶.۱۰ نمونه‌کد ناسازگار (*notify_one()*) ..... ۲۹۲
۲.۶.۱۰ راه‌حل سازگار (*notify_all()*) ..... ۲۹۴
۳.۶.۱۰ راه‌حل سازگار (استفاده از *notify_one()* با یک متغیر شرطی منحصر به‌فرد برای هر نخ) ..... ۲۹۵
۴.۶.۱۰ ارزیابی خطر ..... ۲۹۶

**۷.۱۰ CON56-CPP یک میوتکس غیربازگشتی را که متعلق به فراخوانی نخ است، قفل نکنید ..... ۲۹۷**

۱.۷.۱۰ نمونه‌کد ناسازگار ..... ۲۹۷
۲.۷.۱۰ راه‌حل سازگار ..... ۲۹۸
۳.۷.۱۰ ارزیابی خطر ..... ۲۹۹

**پیوست الف: ارزیابی‌های خطر ..... ۳۰۰**

**مراجع ..... ۳۰۴**

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

www.SourceSara.com
سایت سورس سرا - آموزش برنامه نویسی

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران

مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران

مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

# فصل اول

# مقدمه

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

## ۱,۱ مقدمه

با گسترش روزافزون نرم‌افزارها روی پلتفرم‌های مختلف، امنیت نرم‌افزار بیش از پیش مورد توجه قرار گرفته‌است. وجود آسیب‌پذیری در یک نرم‌افزار می‌تواند اهداف یک مهاجم را تحقق بخشد؛ این اهداف می‌توانند شامل دسترسی به اطلاعات شخصی یک فرد در رایانه، دسترسی به شبکه‌ی خصوصی یک شرکت، سرقت اطلاعات موجود در تلفن همراه، مختل‌سازی یک سرور و صدها هدف دیگر باشند.

شاید یک برنامه‌ای به‌ظاهر، کم‌اهمیت و ساده جلوه داده شود اما وجود آسیب‌پذیری در آن ممکن است به خطری جدی برای کاربران آن تبدیل شود. زیرا وجود آسیب‌پذیری مذکور، به تنهایی برای برآورده ساختن اهداف یک مهاجم کافی است. از این‌روی، برنامه‌نویسی ایمن یک برنامه در هر سطح و ابعادی، امری ضروری به‌شمار می‌رود.

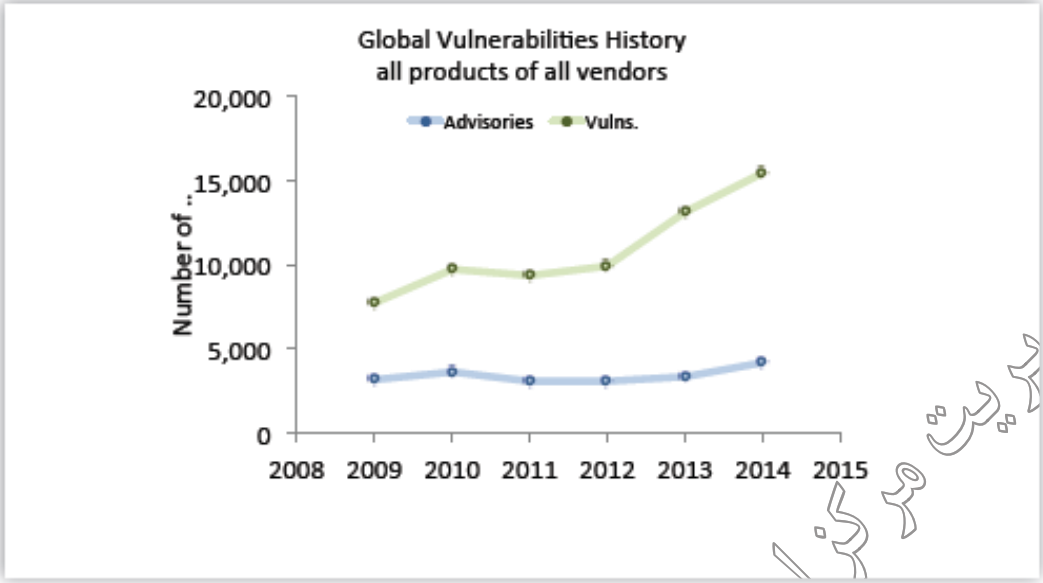
با توجه به اهمیت امنیت نرم‌افزار، شرکت‌های بزرگ دنیا به ارائه‌ی راهکارهایی همچون طراحی زبان‌ها و محیط‌های برنامه‌نویسی، مفسر، و مترجم‌هایی با قابلیت‌های کنترل امنیتی و نیز، راهکارهای امنیتی روی سیستم‌عامل و بسیاری از راهکارهای دیگر پرداخته‌اند، اما با توجه به عدم توانایی راهکارها در کنترل تمامی موارد امنیتی، عدم امکان پیاده‌سازی راهکارهای امنیتی روی برخی ساختارها، ایجاد محدودیت برای دسترسی به برخی منابع و امکانات، و سایر مشکلات کوچک و بزرگ موجود، برنامه‌نویسی امن یک برنامه، بهترین راهکار برای محافظت از آن برنامه به‌شمار می‌رود.

یکی از زبان‌هایی که علاوه بر محبوبیت فراوان در میان برنامه‌نویسان، همیشه از زبان‌های پر بحث در برنامه‌نویسی ایمن بوده‌است، خانواده‌ی زبان C، و به‌خصوص در میان آنها، C++ است. در این زبان‌ها، عمده مدیریت منابع به برنامه‌نویس واگذار شده‌است، که در صورت عدم مدیریت درست آنها، آسیب‌پذیری‌های مختلفی رخ می‌دهد.

## ۲,۱ رشد آسیب‌پذیری نرم‌افزارها

طبق گزارش‌های ارائه‌شده، آسیب‌پذیری‌ها در ۵ سال اخیر، رشد ۵۵ درصدی داشته‌اند. همچنین، آسیب‌پذیری‌های گزارش‌شده، در سال ۲۰۱۴ نسبت به سال ۲۰۱۳، با رشد ۱۱ درصدی همراه بوده‌اند.

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای




این آمار بیان‌گر روند صعودی تعداد آسیب‌پذیری‌های گزارش‌شده در سال‌های اخیر است، که البته رشد کمّی نرم‌افزارها نیز در این بازه، قابل‌توجه است. بدیهی است که این روند افزایشی، زنگ خطری برای کاربران نرم‌افزارها محسوب می‌شود.

برنامه‌های نوشته‌شده به زبان C/C++ از جایگاه ویژه‌ای در این آمار برخوردارند، زیرا این زبان‌ها جدا از محبوبیت، در اکثر محیط‌های برنامه‌نویسی و مفسرها، کنترل کامل روند اجرایی یک برنامه و تخصیص منابع را به برنامه‌نویس سپرده‌اند، که در صورت عدم مدیریت صحیح این موارد، برنامه با مخاطرات امنیتی جدی روبه‌رو خواهد شد. در این میان، نیاز است که برنامه‌نویسان با این مخاطرات آشنا شوند و برنامه را بر مبنای اصول امنیتی طراحی کنند.

## ۳,۱ ضرورت برنامه‌نویسی امن

همان‌طور که اشاره شد، راهکارهای بسیاری برای ایجاد امنیت نرم‌افزار ارائه شده‌است. همچنین، شرکت‌های بزرگ، بخش خاصی را جهت آزمودن سطح امنیتی نرم‌افزار و نیز برقراری امنیت آن بنا نهاده‌اند و یا با تکیه بر مکانیزم برون‌سپاری، نرم‌افزار را به شرکت‌های دیگری، که مجری چنین خدماتی هستند، واگذار نموده‌اند. در این‌راستا، برنامه‌ها توسط ابزارهایی خاص مورد سنجش امنیتی قرار می‌گیرند تا در صورت وجود آسیب‌پذیری، مالکان را مطلع گردانند. اما، هیچ‌یک از راهکارهای فوق به‌طور کامل کارآمد نیستند.

بهترین راهکار برای جلوگیری از بروز آسیب‌پذیری نرم‌افزارها، برنامه‌نویسی پدافندی و ایمن مبنایی آن نرم‌افزار است. امروزه این روش، از ارکان اصلی شرکت‌های نرم‌افزاری آموزش برنامه‌نویسی پدافندی

و ایمن به برنامه‌نویسان به‌شمار می‌رود. همچنین، شرکت‌های ارائه‌دهنده‌ی راهکارهای امنیتی، سرویس‌های آموزش برنامه‌نویسی پدافندی و ایمن را به برنامه‌نویسان و شرکت‌های نرم‌افزاری ارائه می‌کنند.

شایان ذکر است، برنامه‌نویسی پدافندی و ایمن یک برنامه تنها بدان معنی نیست که از بروز آسیب‌پذیری‌ها جلوگیری به عمل آورد، بلکه، در کنار توجه به فاکتورهای امنیتی، راهکاری برای نوشتن یک برنامه بر مبنای اصول برنامه‌نویسی است. حتی در صورتی‌که یک فاکتور به‌صورت مستقیم با آسیب‌پذیری مرتبط نباشد و یک خطر امنیتی محسوب نشود، رعایت کدنویسی امن مهم تلقی می‌گردد.

## ۴.۱ راهنمای سند مورد مطالعه

سند مورد مطالعه، قوانین را جهت کدنویسی امن در C++ تبیین می‌کند. هدف اصلی این قوانین، توسعه‌ی ایمن، قابل اعتماد و امن سیستم‌ها از طرق مختلف، مانند حذف رفتارهای تعریف‌نشده‌ای که می‌توانند منجر به برنامه‌های ناخواسته و نقاط آسیب‌پذیر قابل سوءاستفاده شوند، است.

تطابق قوانین کدنویسی تعریف‌شده در این سند، جهت تأمین و تضمین ایمنی، اعتمادپذیری، و امنیت سیستم‌های نرم‌افزاری توسعه‌یافته در زبان برنامه‌نویسی C++، لازم هستند اما کافی خیر. همچنین، باید یک طراحی امن و ایمن وجود داشته باشد. معمولاً، سیستم حساس به ایمنی، دارای نیازمندی‌های دقیق‌تر و سختگیرانه‌تری نسبت به استانداردهای کدنویسی است. به‌عنوان مثال، در این سیستم‌ها نیاز است که تمامی حافظه به‌صورت ایستا تخصیص داده شود. اگرچه، این استاندارد کدنویسی در سیستم‌های با کیفیت بالایی که قابل اعتماد، مستحکم و پایدار در برابر حملات هستند، کاربرد دارند.

هر قانون شامل یک عنوان، یک توصیف، نمونه‌کدهای ناسازگار، و راه‌حل‌های سازگار است. عنوان، یک اختصار است، اما گاهی به‌تنهایی قادر است کل مفهوم قانون را توصیف نماید. توصیف، نیازمندی‌های اصولی قانون را مشخص می‌کند. نمون کدهای ناسازگار، مثال‌هایی از کد هستند که نقضی را در قانون ایجاد می‌کنند. راه‌حل‌های سازگار نیز، کدهای مشابهی هستند که هیچ قانونی را در این استاندارد کدنویسی نقض نمی‌کنند.

همان‌گونه که پیش از این مطرح شد، استاندارد کدنویسی مستند و اِعمال‌پذیر، مؤلفه‌ای ضروری از کدنویسی در زبان C++ به‌شمار می‌رود. استانداردهای کدنویسی، برنامه‌نویسان را ترغیب می‌سازند تا به جای آن‌که بر اساس تمایلات و ترجیحات شخصی خود عمل نمایند، از مجموعه قوانین یکنواخت مشخص‌شده‌ی مبتنی بر نیازمندی‌های پروژه و نیز سازماندهی آن پیروی کنند. وقتی این امر یک‌بار انجام

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

پذیرد، استانداردهای مذکور می‌توانند با استفاده از فرآیندهای دستی یا خودکار، به‌عنوان معیاری جهت ارزیابی کدهای منبع ارزیابی شوند.

## ۵.۱ کاربرد

هدف اصلی قوانین مورد مطالعه در این مستند، بهبود امنیت نرم‌افزار از طریق بهبود دانش، آگاهی، شیوه‌ها، کاربردها، و ابزار مورد استفاده‌ی توسعه‌دهندگان نرم‌افزار است. این استاندارد می‌تواند توسط پروژه‌ها و سازمان‌ها، جهت توسعه‌ی استانداردهای کدنویسی دنباله‌دار، استفاده شود و روشی مستمر را برای توسعه‌ی امنیت نرم‌افزار فراهم آورد. ممکن است قوانین مذکور توسط قوانین درون‌سازمانی گسترش یابند. این در حالی است که، قوانین مورد مطالعه در این مستند باید مطیع و منعطف باشند تا با استاندارد موردنظر تطابق یابند.

## ۶.۱ شناسه‌ها

سند مورد مطالعه دارای سه شناسه است که عبارتند از: نمونه‌کدهای ناسازگار و راه‌حل‌های سازگار؛ استثنائات؛ و ارزیابی خطر. در ادامه، هر یک از موارد مذکور بررسی خواهند شد.

### ۱.۶.۱ رفتار تعریف‌نشده

رفتار تعریف‌نشده[۱] عبارت است از وقوع اتفاقاتی خارج از انتظار برنامه‌نویس، در برنامه که باعث می‌شود برنامه رفتاری غیرمطلوب از خود نشان دهد. ممکن است رفتار تعریف‌نشده شامل مختل شدن عملکرد برنامه[۲]، خروجی نامربوط و نادرست، بروز آسیب‌پذیری‌های نرم‌افزاری، و سایر موارد باشد.

زبان C++ از زبان‌هایی است که به دلیل آزادی عمل بسیار برنامه‌نویس در آن و عدم وجود سازوکارهای کنترلی و پیش‌گیری درون مفسرهای رایج به‌صورت پیش‌فرض، همیشه در معرض خطر رفتار تعریف‌نشده قرار دارد. وجود رفتار نامتعارف در یک برنامه نه‌تنها امنیت خود آن برنامه، بلکه ممکن است امنیت سیستم‌عامل و شبکه را نیز به خطر بیندازد. جلوگیری از بروز رفتارهای تعریف‌نشده، مقابله با آنها، و نیز کنترل آنها از جمله مباحث مهم برنامه‌نویسی تدافعی و ایمن به‌شمار می‌روند.

ممکن است رفتار نامتعارف به شکل‌های گوناگونی رخ دهند. در این میان، برخی از آن‌ها رایج است و موردبررسی قرار می‌گیرد، اما برخی دیگر از رفتارهای نامتعارف مربوط به اشتباهات برنامه‌نویسی

---

[۱] Undefined behavior

[۲] Crash

مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

برنامه‌نویس است که در هر برنامه و برای هر برنامه‌نویس به‌صورت متفاوت رخ می‌دهد، لذا، برنامه‌نویس باید با ارائه‌ی راهکارهای کلی، این راهکارها را تعمیم دهد، در برنامه‌ی خود از آنها استفاده نماید، و در زمان برنامه‌نویسی به نکات مطرح‌شده توجه داشته‌باشد.

### ۲,۶,۱ نمونه‌کدهای ناسازگار و راه‌حل‌های سازگار

نمونه‌کدهای ناسازگار، کدی را نشان می‌دهند که خط‌مشی موردبحث را نقض می‌نماید. شایان ذکر است که تمامی این موارد تنها مثال هستند، و حذف تمامی موارد از مثال لزوما بدان معنا نیست که کد درحال آنالیز، اکنون سازگار با خط‌مشی مذکور است.

معمولا، نمونه‌کدهای ناسازگار با راه‌حل‌های سازگاری همراه هستند. این راه‌حل‌ها نشان می‌دهند که چگونه یک کد ناسازگار می‌تواند به یک شیوه‌ی امن و سازگار ثبت شود.

### ۳,۶,۱ استثنائات

ممکن است هر قانون، حاوی مجموعه‌ی کوچکی از استثنائات باشد. این استثنائات قادرند جزئیات شرایط مرتبط با خط‌مشی موردبحثی را، که لزومی ندارد ایمنی، اعتمادپذیری، و امنیت نرم‌افزار را تضمین کند، مطرح نمایند. استثنائات، صرفا جهت اطلاع به کار می‌روند و لزومی ندارد که حتما از آنها پیروی شود.

### ۴,۶,۱ ارزیابی خطر

هر خط‌مشی در سند مورد مطالعه، دربردارنده‌ی بخشی به نام ارزیابی خطر است. این بخش سعی دارد تا توسعه‌دهندگان نرم‌افزار را از عواقب ناشی از عدم اجرای صحیح قوانین، مطلع سازد. ممکن است گروه توسعه، از این اطلاعات در راستای تعیین اولویت ترمیم نقوض قانون استفاده نمایند. اولویت با پروژه‌های اصلاح است. عموما، فرض می‌شود که کد جدید به‌گونه‌ای توسعه داده می‌شود که با کل استاندارد کدنویسی و نیز توصیه‌های کاربردی، سازگار باشد.

هر قانون دارای سه اولویت است. اولویت‌ها با استفاده از معیاری مبتنی بر حالت شکست، تخصیص داده می‌شوند. سه مقدار با مقیاس‌های ۱ تا ۳، به معیارهای شدت/دقت[۳]، احتمال[۴]، و هزینه‌ی اصلاح در هر قانون، تخصیص داده می‌شود.

---

[۳] Severity

[۴] Likelihood

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

- **شدت/دقت**: عواقب ناشی از نادیده گرفتن قانون، تا چه میزان جدی خواهد بود؟

| مقدار | معنا | نمونه‌ی آسیب‌پذیری |
|---|---|---|
| ۱ | پایین | حمله‌ی انکار خدمات، خاتمه‌ی غیرعادی |
| ۲ | متوسط | نقض جامعیت داده‌ها، افشای ناخواسته‌ی اطلاعات |
| ۳ | بالا | اجرای کد دلخواه |

- **احتمال**: با چه احتمالی، نقص ناشی از نادیده گرفتن قانون می‌تواند منجر به آسیب‌پذیری قابل سوءاستفاده شود؟

| مقدار | معنا |
|---|---|
| ۱ | بعید |
| ۲ | محتمل |
| ۳ | به احتمال زیاد |

- **هزینه‌ی اصلاح**: هزینه‌ی سازگار نمودن با قانون به چه میزان است؟

| مقدار | معنا | تشخیص | تصحیح |
|---|---|---|---|
| ۱ | بالا | دستی | دستی |
| ۲ | متوسط | خودکار | دستی |
| ۳ | پایین | خودکار | خودکار |

سپس، برای هر قانون، مقادیر در یکدیگر ضرب می‌شوند. حاصل‌ضرب به‌دست‌آمده، مقداری را نتیجه می‌دهد که می‌تواند در اولویت‌بخشی کاربرد قوانین، مورد استفاده قرار گیرد. محدوده‌ی حاصل‌ضرب بین ۱ تا ۲۷، متغیر است. اگرچه، تنها امکان وجود ۱۰ مقدار میسر است: ۱، ۲، ۳، ۴، ۶، ۸، ۹، ۱۲، ۱۸، و ۲۷. قوانین و توصیه‌های حاوی اولویت در محدوده‌های متفاوت، قوانین سطوح مختلفی را تشکیل می‌دهند:

- ۱ الی ۴: قوانین سطح ۳
- ۶ الی ۹: قوانین سطح ۲
- ۱۲ الی ۲۷: قوانین سطح ۱

| سطح | اولویت | تفسیر ممکن |
|---|---|---|

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

| | | |
|---|---|---|
| L1 | ۱۲، ۱۸، ۲۷ | شدت بالا، احتمال زیاد، ارزان جهت ترمیم |
| L2 | ۶، ۸، ۹ | شدت متوسط، محتمل، هزینه‌ی ترمیم متوسط |
| L3 | ۱، ۲، ۳، ۴ | شدت پایین، بعید، گران جهت ترمیم |

در فصول بعدی، قوانین مختلف مرتبط با کدنویسی امن در C++ مطالعه و بررسی می‌شوند. بنابر آنچه که پیش از این مطرح گردید، هر قانون از بخش‌های عناوین، توصیفات، نمونه‌کدهای ناسازگار، راه‌حل‌های سازگار، استثنائات، و ارزیابی خطر تشکیل شده‌است، که به‌طور مفصل شرح داده خواهند شد.

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

# فصل دوم

# اعلان‌ها و مقداردهی اولیه (DCL)

## ۱.۲ DCL50-CPP یک تابع variadic سَبک C تعریف نکنید

می‌توان توابع را به‌گونه‌ای تعریف کرد تا مؤلفه‌های رسمی قوی را در فراخوانی منبعی، که توسط عبارت اعلان پارامتر مشخص می‌شود، بپذیرند. چنین توابعی، توابع variadic نام دارند. در واقع، توابع variadic توابعی هستند که تعداد نامحدودی از پارامترها را دریافت می‌کنند. اعلان یک تابع variadic از نماد سه نقطه، که بیانگر حذف به قرینه‌ی لفظی است (...)، بهره می‌جوید. به‌عنوان مثال، *int printf (const char* format, ...)* نمونه‌ای از اعلان یک تابع variadic است. امکانات کتابخانه‌ای این توابع عبارتند از:

- va_start: دسترسی به پارامترهای تابع variadic را مقدور می‌سازد.
- va_arg: دسترسی به پارامترهای بعدی تابع variadic را مقدور می‌سازد.
- va_copy: یک نسخه‌ی کپی از پارامترهای این تابع تهیه می‌کند.
- va_end: پیمایش پارامترهای این تابع را خاتمه می‌بخشد.
- va_list: نوع این تابع را مشخص می‌کند و اطلاعات مورد نیاز را از طریق امکانات کتابخانه‌ای فوق مقدور می‌سازد.

توابع variadic منعطف هستند، زیرا شمار متعدد و متنوعی از مؤلفه‌های متفاوت را می‌پذیرند. اگرچه، این توابع می‌توانند خطرناک نیز باشند. یک تابع variadic که از حذف به قرینه‌ی سبک C استفاده می‌کند، که از این پس در این گزارش با عنوان تابع variadic سبک C از آن یاد می‌شود، فاقد مکانیزمی جهت بررسی نوع ایمنی مؤلفه‌هایی که از تابع می‌گذرند، و یا تعداد مؤلفه‌هایی که با مفاهیمی از تعریف تابع تطابق دارند است. در نتیجه، یک فراخوان زمان اجرا به تابع variadic سبک C، که مؤلفه‌های نامناسب را از خود عبور می‌دهد، منجر به رفتار تعریف‌نشده‌ای می‌شود. چنین رفتار تعریف‌نشده‌ای می‌تواند جهت اجرای کدهای دلخواه، مورد سوءاستفاده قرار گیرد.

توابع variadic سبک C را تعریف نکنید. اعلان چنین تابعی، که هرگز تعریف نشده، مجاز است. زیرا، مضر نیست و می‌تواند در مضامین ارزیابی‌نشده مفید باشد.

می‌توان با استفاده از توابع variadic، توسط گروهی از پارامترهای تابعی تحت شرایطی که لازم است تعداد متغیری از مؤلفه‌ها به تابع گذر کنند، از معضلات ناشی از توابع variadic سبک C ممانعت به‌عمل

مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

آورد. همچنین، تابع کیورینگ[۵] می‌تواند به‌عنوان جایگزینی برای توابع variadic مورد استفاده قرار گیرد. منظور از تابع کیورینگ، مکانیزمی است که تابعی با تعداد متعددی پارامتر را دریافت می‌کند و توابع متعددی را، که هر یک حاوی تنها یک پارامتر هستند، تولید می‌نماید. به‌عنوان مثال، در مقایسه با توابع خانواده‌ی *printf()* در زبان C، خروجی C++ توسط عملگرهای *std::cout::operator<<()* تک‌مؤلفه‌ای سرباری اجرا می‌شود.

### ۱.۱.۲ نمونه‌کد ناسازگار

در این نمونه‌کد ناسازگار، از یک تابع variadic سبک C جهت افزودن مجموعه‌ای از متغیرهای نوع صحیح[۶] (*int*) به یکدیگر استفاده می‌شود. تابع، مؤلفه‌ها را فرا می‌خواند تا مقدار صفر را بیابد. فراخوانی این تابع بدون گذر از مقدار صفر، به‌عنوان مؤلفه‌ای پس از دو پارامتر اول، منجر به رفتار تعریف‌نشده‌ای می‌شود. شایان ذکر است، گذر از هر نوع دیگری غیر از یک *int* نیز، رفتار تعریف‌نشده‌ای را در پی خواهد داشت.

```
#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;
}
```

### ۲.۱.۲ راه‌حل سازگار (توسعه‌ی بسته‌ی بازگشتی)

در راه‌حل سازگار، یک تابع variadic، که از بسته‌ی پارامتر تابع استفاده می‌کند، جهت اجرای تابع *add* استفاده می‌شود و امکان رفتاری منطقی را برای تمامی منابع فراهم می‌آورد. برخلاف تابع variadic سبک C، که در نمونه‌ی ناسازگاری پیشین استفاده شد، در صورتی‌که لیست پارامترها به صفر ختم شود، راه‌حل سازگار مذکور منجر به رفتار تعریف‌شده‌ای نمی‌شود. به‌علاوه، اگر هر مقداری که از تابع گذر می‌کند از نوع صحیح نباشد، کد به جای آن‌که رفتاری غیر مترقبه از خود نشان دهد، بد فرم[۷] می‌شود.

---

[۵] Currying function

[۶] Integer

[۷] Ill-formed

مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

```
#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...);
}
```

راه‌حل سازگار فوق از *std::enable_if* استفاده می‌کند تا تضمین نماید استفاده از هر مقدار مؤلفه‌ی غیرصحیح، منجر به بدفرم شدن برنامه می‌شود و رفتار نامعینی را به‌همراه نخواهد داشت.

### ۳.۱.۲ راه‌حل سازگار (گسترش لیست آغازگر آماده)

یک راه‌حل سازگار جایگزین، که به توسعه‌ی بازگشتی از مجموعه پارامتر تابع نیازی ندارد، این مجموعه پارامتر تابع را، به‌عنوان بخشی از لیست آغازگر آماده، به لیستی از مقادیر، توسعه می‌دهد. به‌دلیل آن‌که تبدیلات باریک در لیست آغازگر آماده مجاز نیستند، ایمنی نوع، با وجود این‌که *std::enable_if* دربردارنده‌ی هیچ‌یک از پارامترهای variadic نیست، حفظ می‌شود.

```
#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;
}
```

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

### ۴,۱,۲ استثنائات

**DCL50-CPP-EX1**: در صورتی‌که تابعی حاوی پیوند خارجی به زبان C باشد، آن تابع مجاز است که به‌عنوان variadic سبک C تعریف شود. به‌عنوان مثال، ممکن است تابع، تعریفی مورد استفاده در کتابخانه‌ی API زبان C باشد که در C++ نیز اجرا شود.

**DCL50-CPP-EX2**: همان‌گونه که پیش از این مطرح گردید، توابع variadic سبک C که اعلان می‌شوند، هرگز مجاز نیستند که تعریف گردند. به‌عنوان نمونه، وقتی عبارت فراخوانی تابع در یک متن ارزیابی‌شده ظاهر می‌شود، مانند پارامتری در یک عبارت *sizeof*، تحلیل سربار جهت تعیین نتیجه‌ی نوع فراخوانی انجام می‌پذیرد، اما به تعریف تابع نیازی نیست. برخی تکنیک‌های فرابرنامه‌نویسی الگو، SFINA را به کار می‌برند تا از اعلان‌های تابع variadic جهت اجرای پرس‌وجوهای[^] نوع زمان-تفسیر استفاده کنند. در ادامه، مثالی از این نوع مطرح می‌شود.

```
template <typename Ty>
class has_foo_function
{
	typedef char yes[1];
	typedef char no[2];

	template <typename Inner>
	static yes& test(Inner *I, decltype(I->foo()) * = nullptr);
	// Function is never defined.

	template <typename>
	static no& test(...);
	// Function is never defined.

	public:
		static const bool value = sizeof (test <Ty> (nullptr)) == sizeof (yes);
};
```

در این مثال، مقدار *value* بر اساس آن‌که کدام سربار *test()* انتخاب می‌شود، مشخص می‌گردد. اعلان *Inner *I*، امکان استفاده از متغیر *I* را در تعیین‌گر *decltype* ممکن می‌سازد، که منجر به اشاره‌گری از نوعی (احتمالا *void*) با مقدار پیش‌فرضی از *nullptr* می‌شود. با این وجود، اگر اعلانی از *Inner::foo()* وجود نداشته‌باشد، تعیین‌گر *decltype*، بدفرم خواهد شد. نوع دیگری از *test()*، تابع کاندیدی جهت تحلیل سربار منطبق بر SFINAE نخواهد بود. در نتیجه، نوع دیگری از تابع variadic سبک C از *test()*، تنها تابع موجود

[^] Queries

در مجموعه‌ی کاندید خواهد بود. هر دو تابع *test()* اعلان می‌شوند، اما هرگز تعریف نمی‌شوند، زیرا تعاریف آنها برای استفاده در یک مضمون عبارتی ارزیابی‌نشده، موردنیاز نیست.

### ۵,۱,۲ ارزیابی خطر

استفاده‌ی نادرست از یک تابع variadic می‌تواند منجر به یک خاتمه‌ی غیرعادی برنامه، افشای ناخواسته‌ی اطلاعات، و یا اجرای کد دلخواه شود.

| قانون | شدت/دقت | احتمال | هزینه‌ی اصلاح | اولویت | سطح |
|---|---|---|---|---|---|
| **DCL50-CPP** | بالا | محتمل | متوسط | P12 | L1 |

مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

## ۲,۲ DCL51-CPP یک شناسه‌ی ذخیره‌شده را اعلان یا تعریف نکنید

شناسه‌ها و نام‌های مشخصه‌ای که ذخیره می‌شوند، عبارتند از: *override*، *final*، *alignas*، *carries_dependency*، *deprecated*، و *noreturn*. هیچ شناسه‌ی دیگری ذخیره نمی‌گردد. اعلان و یا تعریف یک شناسه در مضمونی که شناسه در آن ذخیره شده‌است، منجر به یک رفتار تعریف‌نشده می‌شود. یک شناسه‌ی ذخیره‌شده را اعلان و یا تعریف نکنید.

### ۱,۲,۲ نمونه‌کد ناسازگار (محافظ سرآیند)

یک روش معمول، استفاده از یک ماکرو در پردازنده‌ای مشروط است که با نفوذ متعدد به فایل سرآیند[۹]، مقابله و از آن محافظت می‌کند. این روش در شرایطی توصیه می‌شود که برنامه‌های متعددی از نام‌های ذخیره‌شده، به‌عنوان محافظان سرآیند[۱۰]، استفاده می‌کنند. ممکن است چنین اسمی هنگام اجرای کتابخانه‌ی الگوی استاندارد C++، با اسامی ذخیره‌شده در سرآیند خود، یا با اسامی ذخیره‌شده‌ای که از پیش به‌طور ضمنی توسط مفسر تعریف شده‌اند، حتی زمانی‌که شامل هیچ سرآیند کتابخانه‌ای استاندارد C++ نمی‌شود، تداخل پیدا کند.

```
#ifndef _MY_HEADER_H_
#define _MY_HEADER_H_

// Contents of <my_header.h>

#endif // _MY_HEADER_H_
```

### ۲,۲,۲ راه‌حل سازگار

راه‌حل سازگار ارائه‌شده در این قسمت، از به‌کارگیری زیرخط[۱۱] در ابتدا و یا انتهای نام محافظ جلوگیری به‌عمل می‌آورد.

```
#ifndef MY_HEADER_H
#define MY_HEADER_H

// Contents of <my_header.h>

#endif // MY_HEADER_H
```

[۹] Header

[۱۰] Header guards

[۱۱] Underscore

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

### ۳،۲،۲ نمونه‌کد ناسازگار (عملگر تحت‌اللفظی تعریف‌شده توسط کاربر)

در این نمونه‌کد ناسازگار، یک عملگر تحت‌اللفظی تعریف‌شده توسط کاربر (*x*"") اعلان می‌شود. اگرچه، به‌پسوند تحت‌اللفظی شناسه‌ها جهت شروع با یک زیرخط نیاز است. پسوندهای تحت‌اللفظی بدون زیرخط پیشوند، برای پیاده‌سازی‌های کتابخانه‌ای آن ذخیره می‌شوند.

```
#include <cstddef>

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

### ۴،۲،۲ راه‌حل سازگار (عملگر تحت‌اللفظی تعریف‌شده توسط کاربر)

در این راه‌حل، عملگر تحت‌اللفظی تعریف‌شده توسط کاربر، *operator""_x* نام دارد، که یک شناسه‌ی ذخیره‌شده نیست.

```
#include <cstddef>
unsigned int operator"" _x(const char *, std::size_t);
```

نام تحت‌اللفظی تعریف‌شده توسط کاربر، *operator""_x* و *_x* نیست. زیرا در غیر این‌صورت، بدان معنا است که برای فضای نام عمومی ذخیره شده‌است.

### ۵،۲،۲ نمونه‌کد ناسازگار (اشیای حوزه‌ی فایل)

در این نمونه‌کد، هر دو اسم اشیای حوزه‌ی فایل (*_max_limit* و *_limit*) با یک زیرخط شروع می‌شوند. به‌دلیل آن‌که نوع آن *static* است، شاید به نظر رسد که اعلان *_max_limit* با اسامی تعریف‌شده توسط اجرا، تصادم نداشته‌باشد. اگرچه، به‌دلیل آن‌که سرآیند <*cstddef*> فراخوانی می‌شود تا *std::size_t* را تعریف کند، پتانسیل تصادم نام وجود دارد. توجه داشته باشید، ممکن است یک مفسر مطابق، اسامی ذخیره‌شده را به‌طور ضمنی، صرف‌نظر از آن‌که آیا هیچ سرآیند کتابخانه‌ای الگوی استاندارد C++ به‌صورت ضمنی فراخوانی شده‌است یا خیر، اعلان کند. به‌علاوه، به‌دلیل آن‌که *_limit* دارای پیوند خارجی است، ممکن است با سمبلی با همان نام تعریف‌شده در کتابخانه‌ی زمان‌اجرا تداخل داشته‌باشد. این امر حتی ممکن است زمانی اتفاق بیفتد که چنین سمبلی در هیچ سرآیندی اعلان نشود. در نتیجه، شروع نام هر شناسه‌ی حوزه‌ی فایل با زیرخط، روشی ایمن نیست. حتی اگر پیوند آن، قابل مشاهده بودن آن را به یک تک واحد ترجمه محدود سازد.

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

```
#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;
 }
```

### ۶.۲.۲ راه‌حل سازگار (اشیای حوزه‌ی فایل)

در این راه‌حل، شناسه‌های حوزه‌ی فایل، با زیرخط شروع نمی‌شوند.

```
#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;
 }
```

### ۷.۲.۲ نمونه‌کد ناسازگار (ماکروی ذخیره‌شده)

در این نمونه‌کد ناسازگار، به‌دلیل آن‌که سرآیند کتابخانه‌ای الگوی استاندارد C++ در <cinttypes> برای فراخواندن <cstdint> مشخص می‌شود، اسم *MAX_SIZE* با ماکرویی از سرآیند <cstdint>، که جهت تعیین حد بالای *std::size_t* استفاده می‌شود، تداخل پیدا می‌کند.

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

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

### ۸.۲.۲ راه‌حل سازگار (ماکروهای ذخیره‌شده)

این راه‌حل، از تعریف مجدد اسامی ذخیره‌شده جلوگیری به عمل می‌آورد.

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

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

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

### ۹.۲.۲ استثنائات

**DCL51-CPP-EX1**: مجاز هستید تا زمانی‌که رفتارهای معنایی یکسان هستند، جهت تطابق با سایر فروشندگان مفسر یا حالات استاندارد زبان، شناسه‌ی مایکرویی ایجاد کنید. حتی ممکن است این شناسه‌ی ماکرویی یک شناسه‌ی ذخیره‌شده باشد.

```
// Sometimes generated by configuration tools such as autoconf
#define const const


// Allowed compilers with semantically equivalent
// extension behavior #define inline __inline
```

**DCL51-CPP-EX2**: مجاز هستید که به‌عنوان یک فروشنده‌ی مفسر یا توسع‌دهنده‌ی کتابخانه‌ی استاندارد، از شناسه‌های ذخیره‌شده برای پیاده‌سازی‌های خود استفاده کنید. شاید شناسه‌های ذخیره‌شده، توسط مفسر در سرآیندهای کتابخانه‌ای استاندارد، یا در سرآیندهایی که توسط سرآیند کتابخانه‌ای واکشی شده‌اند، تعریف شوند. به‌عنوان نمونه، در مثال زیر، اجرای *libc++ STL* اعلان شده‌است.

```
// The following declaration of a reserved identifier exists
// in the libc++ implementation of std::basic_string as a
// public member. The original source code may be found at:
// http://llvm.org/svn/llvm-project/libcxx/trunk/include/string
template<class charT, class traits = char_traits<charT>,
        class Allocator = allocator<charT>>
class basic_string
{
        // ...
        bool __invariants() const;
};
```

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه‌ای

### ۱۰.۲.۲ ارزیابی خطر

استفاده از شناسه‌های ذخیره‌شده می‌تواند منجر به عملکرد نادرست برنامه شود.

| قانون | شدت/دقت | احتمال | هزینه‌ی اصلاح | اولویت | سطح |
|---|---|---|---|---|---|
| DCL51-CPP | پایین | بعید | پایین | P3 | L3 |

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

## ۳,۲ DCL52-CPP هرگز مرجع را از نوع const یا volatile توصیف نکنید

زبان C++ به شما اجازه نمی‌دهد که مقدار نوع مرجع را تغییر دهید. به‌طور مؤثر، با تمامی مراجع به‌گونه‌ای رفتار می‌کند که در قالب *const*، واجد شرایط باشند. یکی از روش‌های مورد استفاده در C++، *cv-qualify* نمودن نوع است. منظور از *cv-qualify* نمودن یک نوع، افزودن *const*، *volatile*، و یا هردوی آنها به نوع موردنظر است. زبان C++ از *cv-qualify* شدن مراجع ممانعت به عمل می‌آورد و تنها امکان *cv-qualify* شدن غیر مرجعات مؤثر است.

وقتی تلاش می‌شود که که نوعی را به‌عنوان بخشی از اعلانی که از نوع مرجع استفاده می‌کند *cv-qualify* نمود، ممکن است برنامه‌نویس به‌صورت تصادفی، عبارت زیر را درج نماید:

```
char &const p;
```

در حالی‌که باید عبارت زیر نوشته شود:

```
char const &p; // Or: const char &p;
```

تلاش نکنید تا یک نوع مرجع را *cv-qualify* نمایید، زیرا به یک رفتار نامشخص منجر می‌شود. برای به چالش کشیدن یک پیغام تشخیص، به یک مفسر منطبق نیاز است. با این‌وجود، اگر مفسر، یک تشخیص مهلک را منتشر نکند، ممکن است برنامه نتایج غیرمنتظره‌ای را تولید نماید؛ به‌عنوان مثال، کاراکتری که توسط *p* ارجاع داده شده‌است، تغییر داده شود.

### ۱,۳,۲ نمونه‌کد ناسازگار

در این نمونه‌کد، یک ارجاع *const-qualified* به یک *char*، به جای ارجاع به یک *const-qualified char*، شکل می‌گیرد. این امر منجر به رفتار تعریف‌نشده‌ای می‌شود.

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

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه‌ای

#### ۱,۱,۳,۲ جزئیات اجرا (مایکروساف ویژوال استودیو)

کد فوق به‌صورت موفقیت‌آمیز با تشخیص اخطار، در برنامه‌ی مایکروسافت ویژوال استودیو[۱۲] (MSVC) نسخه‌ی ۲۰۱۵ تفسیر می‌شود.

```
warning C4227: anachronism used : qualifiers on reference are ignored
```

نتیجه‌ی اجرای کد، خروجی زیر خواهد بود.

```
p
```

#### ۲,۱,۳,۲ جزئیات اجرا (Clang)

با استفاده از برنامه‌ی Clang 3.9، کد، یک تشخیص مهلک تولید می‌کند.

```
error: 'const' qualifier may not be applied to a reference
```

### ۲,۳,۲ نمونه‌کد ناسازگار

این نمونه‌کد، به‌صورت صحیح *p* را به مرجع *const-qualified char* اعلان می‌کند. نتیجه‌ی تغییر *p*، منجر به بدفرم شدن برنامه خواهد شد.

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

### ۳,۳,۲ راه‌حل سازگار

این راه‌حل، توصیف‌گر (صفت) *const* را حذف می‌کند.

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

[۱۲] Microsoft Visual Studio

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه‌ای

### ۴,۳,۲ ارزیابی خطر

ممکن است یک نوع مرجع *const* یا *volatile* به جای تشخیص مهلک، منجر به رفتار تعریف‌نشده شود و مقادیر غیر منتظره‌ی ذخیره‌شونده و نیز نقض جامعیت داده‌های محتمل را به بار آورد.

| قانون | شدت/دقت | احتمال | هزینه‌ی اصلاح | اولویت | سطح |
|---|---|---|---|---|---|
| DCL52-CPP | پایین | بعید | پایین | P3 | L3 |

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

## ۴.۲ DCL53-CPP اعلاناتی را که از نظر نحوی مبهم هستند، ننویسید

ممکن است دستوری ایجاد کنید که در تفسیر آن ابهام وجود داشته‌باشد. در واقع، ممکن است که یا به‌عنوان یک عبارت صرف، تفسیر شود و یا به‌عنوان یک اعلان. این نوع از دستور، تجزیه‌ی رأس[۱۳] نامیده می‌شود، زیرا مفسر باید از قوانین ابهام‌زدایی جهت تعیین نتایج معنایی استفاده کند.

یک تجزیه‌ی رأس مشابه در مضمون اعلان موجود است، جایی که دستور می‌تواند به‌صورت ابهام‌آمیزی به‌عنوان اعلان تابع و یا مقداردهنده‌ی اولیه تفسیر شود.

هرگز اعلانی را که از نظر ظاهری مبهم است، ننویسید. با ظهور دستور مقداردهی نخستین یکپارچه از طریق یک braced-init-list، می‌توان اعلان را از یک عبارت صرف تمییز داد. اعلان‌ها می‌توانند توسط قالب‌های غیرتابعی، با مقداردهی اولیه از طریق "="، و یا با حذف پرانتزهای اضافی پیرامون نام پارامترها، ابهام‌زدایی شوند.

### ۱.۴.۲ نمونه‌کد ناسازگار

در این نمونه‌کد، انتظار می‌رود یک متغیر محلی ناشناس از نوع *std::unique_lock*، میوتکس[۱۴] ملقب به *m* را از طریق درستی *RAII*، قفل کند و بگشاید. اگرچه، اعلان از نظر نحوی مبهم است، زیرا می‌تواند به‌عنوان اعلان یک شی ناشناس و فراخوانی سازنده‌ی مبدل تک‌پارامتری آن تفسیر شود. دستور استفاده‌شده در این مثال، دومین را به جای نخستین تعریف می‌کند. بنابراین، شی میوتکس هرگز قفل نمی‌شود.

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

### ۲.۴.۲ راه‌حل سازگار

در این راه‌حل، شناسه‌ای (غیر از *m*) به‌عنوان شی قفل در نظر گرفته می‌شود و سازنده‌ی مبدل مناسب، فراخوانی می‌شود.

---

[۱۳] Vertex pars

[۱۴] Mutex

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

### ۳,۴,۲ نمونه‌کد ناسازگار

در این نمونه‌کد، تلاشی برای اعلان یک متغیر محلی از نوع *Widget*، به نام *w* صورت می‌پذیرد. این در حالی است که سازنده‌ی پیش‌فرض در حال اجرا است. اگرچه، ظاهرا این اعلان از نقطه‌نظر نحوی دارای ابهام است. در واقع، ممکن است اعلانی از یک اشاره‌گر تابع، که هیچ پارامتری را نمی‌پذیرد و یک *Widget* را باز می‌گرداند، به‌عنوان اعلانی از یک متغیر محلی نوع *Widget* در نظر گرفته شود. دستور استفاده‌شده در این مثال، پیشین را به جای پسین تعریف می‌کند.

```
#include <iostream>
struct Widget
{
	Widget()
	{
		std::cout << "Constructed" << std::endl;
	}
};
void f()
 {
	Widget w();
}
```

در نتیجه، این برنامه هیچ خروجی را تفسیر و یا چاپ نمی‌کند. زیرا، سازنده‌ی پیش‌فرض هرگز به‌صورت حقیقی فراخوانی نمی‌شود.

### ۴,۴,۲ راه‌حل سازگار

این راه‌حل، دو راه‌حل سازگار یکسان را جهت نوشتن اعلان، ارائه می‌دهد. روش اول، پرانتزهای پس از اعلان متغیر را حذف می‌کند. این امر تضمین می‌کند که دستور، به جای اعلان تابع، به اعلان متغیر مرتبط است. روش دوم نیز از یک braced-init-list برای مقداردهی نخستین مستقیم متغیر محلی استفاده می‌کند.

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

```
#include <iostream>
struct Widget
{
	Widget()
	{
		std::cout << "Constructed" << std::endl;
	}
};
void f()
 {
	Widget w1; // Elide the parentheses
	Widget w2{}; // Use direct initialization
}
```

اجرای این برنامه، دو مرتبه خروجی *Constructed* را تولید می‌کند: یک بار برای *w1*؛ و بار دیگر برای *w2*.

### ۵,۴,۲ نمونه‌کد ناسازگار

این نمونه‌کد، یک تجزیه‌ی آزاردهنده[۱۵] را نشان می‌دهد. اعلان *Gadget g (Widget (i))*، به‌عنوان اعلان یک شی *Gadget* تک‌پارامتری تحلیل نمی‌شود. بلکه، به‌عنوان اعلان تابع با مجموعه‌ی افزونه‌ای از پرانتزها، حول یک پارامتر تحلیل می‌گردد.

```
#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;
}
```

[۱۵] Vexing parse

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

پرانتزهای پیرامون اسامی پارامترها، اختیاری هستند. بنابراین، کد زیر یک املای یکسان معنایی از اعلان است.

```
Gadget g(Widget i);
```

در نتیجه، این برنامه، برنامه‌ی ساختاریافته‌ای به‌شمار می‌رود و تنها ۳ را به‌عنوان خروجی چاپ می‌کند. زیرا، هیچ شی *Gadget* یا *Widget* ساخته نمی‌شود.

### ۶,۴,۲ راه‌حل سازگار

این راه‌حل، دو روش سازگار یکسان را برای نوشتن اعلان *g* نشان می‌دهد. نخستین اعلان، *g1*، از یک مجموعه‌ی اضافی از پرانتزهای حول پارامتر، برای فراخوان سازنده استفاده می‌کند، که مفسر را مجبور می‌سازد آن را، به جای آن‌که به‌عنوان تابع تحلیل نماید، به‌عنوان متغیری محلی از نوع *Gadget* تجزیه کند. اعلان دوم، *g2*، از مقداردهی اولیه‌ی سیستم برای ایجاد تأثیری مشابه استفاده می‌کند.

```
#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;
}
```

اجرای برنامه‌ی فوق، خروجی مطلوب و موردنظر را تولید می‌نماید.

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

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

### ۷.۴.۲ ارزیابی خطر

اعلانات مبهم نحوی می‌توانند منجر به اجرای ناخواسته‌ی برنامه شوند. اگرچه، احتمالا تست مقدماتی، نقض‌های این قانون را مشخص خواهد کرد.

| قانون | شدت/دقت | احتمال | هزینه‌ی اصلاح | اولویت | سطح |
|---|---|---|---|---|---|
| DCL53-CPP | پایین | بعید | متوسط | P2 | L3 |

مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

## ۵.۲ DCL54-CPP توابع تخصیص و آزادسازی را به‌عنوان زوجی در یک حوزه، سربار کنید

توابع تخصیص و آزادسازی می‌توانند در هر دو حوزه‌ی عمومی و کلاس دچار سربار شوند. اگر یک تابع تخصیص در یک حیطه‌ی مشخص دچار سربار شود، تابع آزادسازی متناظر نیز باید در همان حوزه دچار سربار شود (و بالعکس).

احتمالا شکست در سربار کردن تابع ذخیره‌سازی پویای متناظر، منجر به نقض قوانینی می‌شود. به‌عنوان مثال، اگر یک تابع سرریزشده، از یک پشته‌ی خصوصی جهت اجرای تخصیصات خود استفاده کند، احتمالا عبور از اشاره‌گری، که توسط آن به تابع آزادسازی پیش‌فرض ارجاع داده شده‌است، رفتار تعریف‌نشده‌ای را در پی خواهد داشت. حتی در چنین شرایطی، که تابع تخصیص به‌صورت نامحدود از تخصیص‌گر پیش‌فرض برای به‌دست آوردن یک اشاره‌گر به حافظه استفاده می‌کند، ممکن است شکست در راستای سربار نمودن یک تابع آزادساز متناظر، به‌علت به‌روزرسانی ننمودن داده‌های داخلی برای تخصیص‌گر مشتری، منجر به ترک ناگهانی و غیرمنتظره‌ی برنامه شود.

قابل قبول است که یک تابع تخصیص و یا آزادسازی حذف‌شده، بدون تابع free store خود تعریف شود. به‌عنوان مثال، مرسوم است که وقتی کلاس، یک جایابی[۱۶] تابع *new* را تعریف می‌کند، یک تابع تخصیص یا آزادسازی جایابی‌نشده‌ی محذوف، به‌عنوان یک تابع عضو کلاس تعریف می‌شود. این امر، از تخصیص یکسان با فراخوانی *new* برای نوع کلاس و یا تخصیص با فراخوانی *delete*، روی اشاره‌گرهای یک شی از نوع کلاس، جلوگیری به عمل آورد. قابل قبول است که اعلان یک تابع تخصیص یا آزادسازی، بدون تابع free store متناظر آن، به دلایل مشابه صورت پذیرد، اما تعریف خیر. اگرچه، تعریف نباید به‌گونه‌ای تعبیه شود که اجازه‌ی دسترسی به تابع free store داخل تابع عضو کلاس را داشته‌باشد.

### ۱.۵.۲ نمونه‌کد ناسازگار

در این نمونه‌کد، یک تابع تخصیص عمومی سربار می‌شود. اگرچه، تابع آزادسازی متناظر آن اعلان نمی‌شود. هرگونه تلاش جهت حذف شی، در جایی‌که شی به‌وسیله‌ی تابع تخصیص سرباری تخصیص داده شده‌است، منجر به رفتاری تعریف‌نشده و نقض قانون آزادسازی پویای منابع تخصیص‌یافته خواهد شد.

[۱۶] Placement

```
#include <Windows.h>
#include <new>
void *operator new(std::size_t size) noexcept(false)
{
        // Private, expandable heap.
        static HANDLE h = ::HeapCreate(0, 0, 0);
         if (h)
        {
                return ::HeapAlloc(h, 0, size);
        }
         throw std::bad_alloc();
 }
// No corresponding global delete operator defined.
```

### ۲.۵.۲ راه‌حل سازگار

در این راه‌حل، تابع آزادسازی متناظر نیز به‌صورت عمومی تعریف خواهد شد.

```
#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);
                        }
                }
};
```

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

```
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);
}
```

### ۳،۵،۲ نمونه کد ناسازگار

عملگر *operator new ()* در حوزه‌ی کلاس دچار سربار می‌شود، اما *operator delete ()* در این حوزه دچار سربار نمی‌شود. برخلاف فراخوانی‌های تابع تخصیص سرباری، که از طریق تابع تخصیص عمومی پیش‌فرض انجام می‌پذیرد، یک شی از نوع S وجود دارد که باید تخصیص داده‌شود. هر تلاش جهت حذف شی، به‌علت شکست در به‌روزرسانی منطبق بر ساماندهی تخصیص، برنامه را در یک حالت نامشخص رها خواهد کرد.

```
#include <new>
extern "C++" void update _bookkeeping (void *allocated_ptr, std::size_t size, bool alloc);
struct S
{
        void  *operator new(std::size_t size) noexcept (false)
        {
                void *ptr = ::operator new (size);
                update_bookkeeping (ptr, size, true);
                return ptr;
        }
} ;
```

### ۴،۵،۲ راه‌حل سازگار

*operator delete ()* متناظر، در همان حوزه‌ی کلاس دچار سربار می‌شود.

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

```
#include <new>
extern "C++" void update _bookkeeping (void *allocated_ptr, std::size_t size, bool alloc);
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);
        }
} ;
```

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

### ۵,۵,۲ استثنائات

**DCL54-CPP-EX1**: ممکن است یک تابع آزادسازی جایابی برای تابع تخصیص متناظر آن حذف شود. اما، این اتفاق تنها در شرایطی رخ می‌دهد که تضمین شود تخصیص جایابی شی و ساخت شی، به‌صورت *noexcept (true)* باشند. به‌دلیل آن‌که توابع آزادسازی جایابی، وقتی مقداردهی اولیه‌ی شی از طریق ایجاد[۱۷] یک استثنا خاتمه می‌یابد، به‌صورت پویا فراخوانی می‌شوند، ایمن است زمانی‌که استثنائات نتوانند ایجاد شوند، تابع آزادسازی جایابی حذف شود. به‌عنوان مثال، برخی فروشنده‌ها، پرچم‌های مفسری را که از غیرفعال‌سازی استثنائات حمایت می‌کنند (مانند استثنائات *-fno-cxx-exceptions* در Clang و *-EHs-c/* در مایکروسافت ویژوال استودیو)، اجرا می‌کنند. این پرچم‌ها دربردارنده‌ی رفتار تعریف‌نشده‌ی اجرایی در زمانی هستند که یک استثنا ایجاد می‌شود. اما، مشابه فراخوانی *abort ()*، به‌صورت عمومی منجر به ختم برنامه می‌شود.

### ۶,۵,۲ ارزیابی خطر

استفاده‌ی ناسازگار از *new* و *delete* می‌تواند منجر به حمله‌ی انکار خدمات شود.

| قانون | شدت/دقت | احتمال | هزینه‌ی اصلاح | اولویت | سطح |
|---|---|---|---|---|---|
| DCL54-CPP | پایین | محتمل | پایین | P6 | L2 |

[۱۷] thrown

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

## ۶,۲ DCL55-CPP از نشت اطلاعات هنگام گذر از یک شی کلاس در حیطه‌ی یک مرز اعتماد، جلوگیری نمایید

بیت‌های پدینگ[۱۸]، نوعی اعداد صحیح بدون علامت هستند که در دسترس کاربر قرار دارند. به‌عنوان مثال، ماشینی را در نظر بگیرید که از یک جفت ۱۶ بیتی کوتاه، که هر یک حاوی بیت علامت خود است، برای ایجاد یک نوع صحیح (*int*) ۳۲ بیتی استفاده می‌کند. در چنین شرایطی، بیت علامت کوتاه پایین‌تر نادیده گرفته می‌شود. بنابراین، به‌عنوان یک نوع صحیح (*int*) ۳۲ بیتی علامت‌دار، یک بیت پدینگ (در میانه‌ی ۳۲ بیت) وجود دارد، که در تعیین مقدار صحیح ۳۲ بیتی علامت‌دار نادیده گرفته می‌شود.

بیت پدینگ می‌تواند برای هر پیامی با هر سایزی به کار گرفته شود. یک تک‌بیت ۱ به پیام اضافه می‌شود و سپس، تعداد صفرهای مورد نیاز بعدی به آن اضافه می‌شوند. شایان ذکر است، ممکن است به هیچ تعداد صفری نیاز نباشد. تعداد صفرهای اضافه‌شونده، به مرز و محدوده‌ی بلاک بستگی دارد. در واقع، هر اندازه که بلاک مجاز است گسترش یابد، به آن صفر اضافه می‌شود. به‌عنوان مثال، یک پیام ۲۳ بیتی که توسط ۹ بیت پُر می‌شوند تا یک بلاک ۳۲ بیتی را تشکیل دهد، در زیر نشان داده شده‌است.

| 1011 1001 1101 0100 0010 0111 0000 0000 |

پدینگ، نخستین گام از یک طرح پدینگ دو مرحله‌ای است که در بسیاری از توابع هَش، از قبیل MD5 و SHA به کار گرفته می‌شود.

ممکن است بیت‌های پدینگ در هر نقطه‌ای داخل یک شی کلاس (به‌عنوان مثال در ابتدای شی، در مواردی که یک فیلد بیت بی‌نام به‌عنوان اولین عضو اعلان‌شده در یک کلاس قرار دارد) ظاهر شوند. البته، در مواردی که با مقدار صفر آغاز شده‌باشند، بیت‌های پدینگ شامل مقادیر تعیین‌نشده‌ای هستند که ممکن است شامل اطلاعات حساس نیز باشند.

هنگامی‌که گذری از یک اشاره‌گر متعلق به یک نمونه شی کلاس در یک مرز اعتماد به حوزه‌ای مطمئن اتفاق می‌افتد، لازم است برنامه‌نویس تضمین نماید که بیت‌های پدینگ این شی شامل اطلاعات حساس نیستند.

[۱۸] Padding

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

### ۱,۶,۲ نمونه‌کد ناسازگار

این نمونه‌کد در فضای هسته اجرا می‌شود و داده‌ها را از *arg* به فضای کاربر کپی می‌کند. اگرچه، ممکن است بیت‌های پدینگ داخل، شی استفاده شوند. به‌عنوان مثال، می‌توان به استفاده از بیت‌های پدینگ جهت تضمین تخصیص مناسب اعضای داده‌های کلاس اشاره نمود. ممکن است بیت‌های پدینگ مذکور حاوی اطلاعات حیاتی باشند که بعداً، هنگامی که داده‌ها به فضای کاربر کپی می‌شوند، صرف‌نظر از آن‌که نحوه‌ی کپی آنها به چه ترتیب است، نشت کنند.

```
#include <cstddef>
struct test
{
        int a;
        char b;
        int c;
};
// Safely copy bytes to user space
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));
}
```

### ۲,۶,۲ نمونه‌کد ناسازگار

در این نمونه‌کد، *arg* به‌صورت مستقیم مقداردهی اولیه می‌شود. به‌دلیل آن‌که *test* فاقد یک ساختار پیش‌فرض کاربرمعابانه است، مقداردهی اولیه توسط یک مقداردهی اولیه‌ی صفر آغاز می‌شود. این روند تضمین می‌کند پیش از آن‌که مقداردهی اولیه‌ای بیشتری صورت گیرد، بیت‌های پدینگ به صفر مقداردهی اولیه می‌شوند. این امر مستلزم استفاده از *std::memset ()* جهت مقداردهی اولیه‌ی تمامی بیت‌های شی به مقدار صفر است.

```
#include <cstddef>
struct test
{
        int a;
        char b;
        int c;
};
// Safely copy bytes to user space
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));
}
```

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه‌ای

با این‌وجود، مفسرها مختار هستند تا *arg.b = 2* را با تنظیم بایت پایین یک ثبات ۳۲ بیتی به ۲ اجرا، بیت‌های بالا را بدون تغییر رها، و تمامی ۳۲ بیت ثبات را در حافظه ذخیره کنند. این امر می‌تواند منجر به نشت بایت‌های بالای موجود در ثبات به کار رفته شود.

### ۳,۶,۲ راه‌حل سازگار

این راه‌حل، ساختار داده‌ها را، پیش از کپی آن به یک مضمون نامطمئن، مرتب می‌کند. این کد تضمین می‌کند که هیچ بیت پدینگ مقداردهی اولیه‌نشده‌ای، برای کاربران غیرممتاز کپی نشود. اکنون، ساختار کپی‌شده به فضای کاربر، یک ساختار بسته‌بندی‌شده است و تابع *copy_to_user ()* باید جهت بازتولید ساخته‌ی پَدشده‌ی اصلی، آن را از حالت بسته خارج نماید.

```
#include <cstddef>
#include <cstring>
struct test
{
	int a;
	char b;
	int c;
};
// Safely copy bytes to user space.
extern int copy_to_user(void *dest, void *src, std::size_t size);
void do_stuff(void *usr_buf)
{
	test arg{1, 2, 3};
	// May be larger than strictly needed.
	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 */);
}
```

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

### ۴,۶,۲ راه‌حل سازگار (بایت‌های پدینگ)

بیت‌های پدینگ می‌توانند منحصرا به‌عنوان فیلدهای داخل ساختار اعلان شوند. این راه‌حل، قابل حمل نیست. زیرا، به اجرا و معماری حافظه‌ی هدف بستگی دارد. راه‌حل زیر، یک نمونه‌ی مشخص به معماری x86-32 است.

اعلان *static_assert ()* یک عبارت ثابت و یک پیغام خطا را می‌پذیرد. عبارت، در زمان تفسیر ارزیابی می‌شود و اگر اشتباه باشد، تفسیر خاتمه می‌یابد و پیغام خطا به‌عنوان تشخیص استفاده می‌شود. درج صحیح بایت‌های پدینگ به *struct* باید تضمین کند که هیچ بایت پدینگ اضافی توسط مفسر اضافه نمی‌شود. در نتیجه، لازم است هر دو اعلان ایستا درست باشند. اگرچه، این مفروضات باید تأیید اعتبار شوند تا تضمین نمایند که راه‌حل برای یک اجرای مشخص، صحیح است.

```
#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) {}
};
// Ensure c is the next byte after the last padding byte.
static_assert(offsetof(test, c) == offsetof(test, padding_3) + 1, "Object contains intermediate padding");
// Ensure there is no trailing padding.
static_assert(sizeof(test) == offsetof(test, c) + sizeof(int), "Object contains trailing padding");
// Safely copy bytes to user space.
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));
}
```

### ۵,۶,۲ نمونه‌کد ناسازگار

در این نمونه‌کد، بیت‌های پدینگی که باید افزایش یابند، شامل موارد زیر هستند:

- بیت‌های پدینگ تخصیص مابعد یک جدول روش مجازی و یا کلاس مبنایی مجازی، جهت تخصیص زیرمجموعه‌ای از عضو داده.
- بیت‌های پدینگ تخصیص، جهت تعیین موقعیت زیرمجموعه‌ای از عضو داده در یک مرز تخصیص‌شده‌ی مناسب.
- بیت‌های پدینگ تخصیص، جهت تعیین موقعیت اعضای داده با سطوح کنترل و دسترسی متنوع.

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

- بیت‌های پدینگ فیلد بیت زمانی، که مجموعه‌ای ترتیبی از فیلدهای بیتی یک واحد تخصیص کامل را پر نمی‌کند.
- بیت‌های پدینگ فیلد بیتی، هنگامی‌که دو فیلد بیتی مجاور با انواع بنیادین متفاوتی اعلان می‌شوند.
- بیت‌های پدینگ زمانی، که یک فیلد بیتی با طولی بیشتر از تعداد بیت‌های واحد تخصیص بنیادین اعلان می‌شود.
- بیت‌های پدینگ، جهت تضمین نمونه کلاسی که به‌طور مناسب برای استفاده در یک آرایه، تخصیص داده خواهد شد.

این کد نمونه، در فضای هسته اجرا می‌شود و داده‌ها را از فضای *arg* به فضای کاربر کپی می‌کند. این در حالی است که، شاید بیت‌های پدینگ داخل نمونه-شی، حاوی اطلاعات حساسی باشند که بعدا، زمانی‌که داده به فضای کاربر کپی شوند، نشت کنند.

```
#include <cstddef>
class base
{
        public:
                virtual ~base() = default;
};
class test : public virtual base
{
        alignas(32) double h;
        char i;
        unsigned j : 80; protected: unsigned k; unsigned l : 4; unsigned short m : 3;
         public:
                char n;
                double o;
                test(double h, char i, unsigned j, unsigned k, unsigned l, unsigned short m, char n, double o) :
                h(h), i(i), j(j), k(k), l(l), m(m), n(n), o(o) {}
                virtual void foo();
};
// Safely copy bytes to user space.
extern int copy_to_user(void *dest, void *src, std::size_t size);
void do_stuff(void *usr_buf)
{
        test arg{0.0, 1, 2, 3, 4, 5, 6, 7.0};
        copy_to_user(usr_buf, &arg, sizeof(arg));
}
```

بیت‌های پدینگ، اجرا-محور هستند. بنابراین، ممکن است طرح این اشیای کلاس میان مفسرها و معماری‌ها، به‌صورت گوناگونی صورت پذیرد. هنگامی‌که GCC 5.3.0 برای x86-32 تفسیر شود، شی *test*، به ۹۶ بایت ذخیره‌سازی جهت تطبیق ۲۹ بایت داده (۳۳ بایت شامل rtable) نیاز دارد و طرحی را، مشابه آنچه که در ادامه مشاهده می‌کنید، دربر می‌گیرد.

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

| Offset (bytes (bits)) | Storage Size (bytes (bits)) | Reason | Offset | Storage Size | Reason |
|---|---|---|---|---|---|
| 0 | 1 (32) | vtable pointer | 56 (448) | 4 (32) | unsigned k |
| 4 (32) | 28 (224) | data member alignment padding | 60 (480) | 0 (4) | unsigned l : 4 |
| 32 (256) | 8 (64) | `double h` | 60 (484) | 0 (3) | unsigned short m : 3 |
| 40 (320) | 1 (8) | `char i` | 60 (487) | 0 (1) | unused bit-field bits |
| 41 (328) | 3 (24) | data member alignment padding | 61 (488) | 1 (8) | char n |
| 44 (352) | 4 (32) | unsigned j : 80 | 62 (496) | 2 (16) | data member alignment padding |
| 48 (384) | 6 (48) | extended bit-field size padding | 64 (512) | 8 (64) | double o |
| 54 (432) | 2 (16) | alignment padding | 72 (576) | 24 (192) | class alignment padding |

### ۶,۶,۲ راه‌حل سازگار

بنابر پیچیدگی ساختار داده، این راه‌حل سازگار، به جای آن‌که تلاش کند تا بایت‌های پدینگ را به‌صورت دستی برای همه محاسبه نماید، داده‌ی شی را، پیش از کپی به مضمون نامطمئن، کپی می‌کند.

```
#include <cstddef>
#include <cstring>
class base
{
        public:
                virtual ~base() = default;
};
 class test : public virtual base
 {
        alignas(32) double h;
        char i;
        unsigned j : 80;
        protected:
                unsigned k;
                unsigned l : 4;
                unsigned short m : 3;
        public:
                char n;
                double o;
                test(double h, char i, unsigned j, unsigned k, unsigned l, unsigned short m, char n, double o) :
                h(h), i(i), j(j), k(k), l(l), m(m), n(n), o(o) {}
                virtual void foo();
                bool serialize(unsigned char *buffer, std::size_t &size)
                {
                        if (size < sizeof(test))
                        {
                                return false;
                        }
```

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

```
                std::size_t offset = 0;
                std::memcpy(buffer + offset, &h, sizeof(h)); offset += sizeof(h); std::memcpy(buffer
                + offset, &i, sizeof(i)); offset += sizeof(i);
                // Only sizeof(unsigned) bits are valid, so the following is not narrowing.
                unsigned loc_j = j;
                std::memcpy(buffer + offset, &loc_j, sizeof(loc_j));
                offset += sizeof(loc_j); std::memcpy(buffer + offset, &k, sizeof(k));
                offset += sizeof(k);
                unsigned char loc_l = l & 0b1111;
                std::memcpy(buffer + offset, &loc_l, sizeof(loc_l));
                offset += sizeof(loc_l);
                unsigned short loc_m = m & 0b111;
                std::memcpy(buffer + offset, &loc_m, sizeof(loc_m));
                offset += sizeof(loc_m);
                std::memcpy(buffer + offset, &n, sizeof(n)); offset += sizeof(n);
                std::memcpy(buffer + offset, &o, sizeof(o));
                offset += sizeof(o); size -= offset; return true;
        }
};
// Safely copy bytes to user space.
extern int copy_to_user(void *dest, void *src, size_t size);
void do_stuff(void *usr_buf)
{
        test arg{0.0, 1, 2, 3, 4, 5, 6, 7.0};
        // May be larger than strictly needed, will be updated by calling serialize() to the size of the buffer
        //remaining.
        std::size_t size = sizeof(arg);
        unsigned char buf[sizeof(arg)];
        if (arg.serialize(buf, size))
        {
        copy_to_user(usr_buf, buf, sizeof(test) - size);
        }
        else
        {
                // Handle error
        }
}
```

این کد تضمین می‌کند که هیچ بیت پدینگ مقداردهی اولیه نشده‌ای، برای کاربر غیرممتاز کپی نشود. اکنون، ساختار کپی‌شده به فضای کاربر، یک ساختار بسته‌بندی‌شده است و تابع *copy_to_user ()* باید جهت خلق مجدد ساختار پدینگ اصلی، آن را از حالت بسته خارج کند.

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

### ۷,۶,۲ ارزیابی خطر

ممکن است بیت‌های پدینگ، غیرعامدانه، شامل داده‌های حساسی مانند اشاره‌گرها به ساختارهای داده‌ای هسته و یا رمزهای عبور باشند. یک اشاره‌گر به چنین ساختاری، باید بتواند به سایر توابع گذر کند. این امر موجب نشت اطلاعات خواهد شد.

| قانون | شدت/دقت | احتمال | هزینه‌ی اصلاح | اولویت | سطح |
|---|---|---|---|---|---|
| DCL55-CPP | پایین | بعید | بالا | P1 | L3 |

*۱,۷,۶,۲ آسیب‌پذیری‌های مرتبط*

آسیب‌پذیری‌های متعددی در هسته‌ی لینوکس از نقض این قانون نشأت می‌گیرند.

CVE-2010-4083 یک آسیب‌پذیری را توصیف می‌کند که فراخوانی سیستمی *semctl ()*، به کاربران غیرممتاز اجازه می‌دهد تا حافظه‌ی پشته‌ی هسته‌ای مقداردهی اولیه‌نشده را بخوانند. زیرا، فیلدهای متنوع از یک *semid_ds struct* اعلان‌شده روی پشته، پیش از آن‌که برای کاربر کپی شوند، دچار تغییر و یا به صفر تنظیم نمی‌شوند.

CVE-2010-3881 یک آسیب‌پذیری را در ساختار پدینگ و فیلدهای ذخیره‌شده در ساختار داده‌ای مشخص در QEMU-KVM، که به‌صورت مناسب پیش از آن‌که در فضای کاربر کپی شوند مقداردهی اولیه نشده‌اند، توصیف می‌کند. یک کاربر ممتاز میزبان، قادر است با دسترسی به */dev/kvm*، از این نقض جهت نشت حافظه‌ی پشته‌ی هسته به فضای کاربر استفاده کند.

CVE-2010-3477 نشت اطلاعات هسته را در act_police توصیف می‌کند. act_police مکانی است که ممکن است ساختارهای مقداردهی اولیه‌شده‌ی نامناسب در کد زباله‌ی کنترل ترافیک، اجازه‌ی افشای حافظه‌ی هسته را به کاربردهای فضای کاربر صادر کند.

مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

## ۷.۲ DCL56-CPP از ایجاد چرخه‌ها در مقداردهی اولیه‌ی اشیای ایستا بپرهیزید

تابع را مجددا طی مقداردهی اولیه‌ی یک اعلان متغیر ایستا، وارد نکنید. در غیر این‌صورت، برنامه رفتار تعریف‌نشده‌ای را از خود بروز خواهد داد. به بازگشت نامتناهی جهت پیگیری رفتار نامشخص نیاز نیست. تنها تابع نیاز دارد یک‌بار به بخشی از مقداردهی اولیه بازگردد. به‌دلیل مقداردهی اولیه‌ی ایمن نخ متغیرها، غالبا یک تک‌فراخوانی بازگشتی منجر به بن‌بستی خواهد شد، که به‌دلیل قفل یک همگام‌سازی غیر بازگشتی اتفاق می‌افتد.

یک مقداردهی اولیه‌ی مستقل، بین اشیای ایستای دارای مقداردهی اولیه‌ی پویا ایجاد ننمایید، مگر آن‌که آنها با توجه به یکدیگر مرتب شده‌باشند. مقداردهی آغازین مرتب، مخصوصا نمونه‌های شایع در محدوده‌ی واحد ترجمه، منجر به رفتار تعریف‌نشده‌ای خواهد شد.

### ۱.۷.۲ نمونه‌کد ناسازگار

این نمونه‌کد سعی دارد که یک تابع فاکتوریل مؤثر را با استفاده از نهان‌سازی[۱۹] اجرا کند. به‌دلیل آن‌که مقداردهی اولیه‌ی آرایه‌ی محلی ایستای *cache* دارای بازگشت است، رفتار تابع تعریف‌نشده‌است؛ حتی اگر بازگشت، نامتناهی باشد.

```
#include <stdexcept>
int fact(int i) noexcept(false)
{
	if (i < 0)
	 {
		// Negative factorials are undefined.
		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;
}
```

[۱۹] Caching

*۱,۱,۷,۲ جزئیات پیاده‌سازی*

در ویژوال استودیوی ۲۰۱۵ و GCC 6.1.0، مقداردهی آغازین بازگشتی *cache* به بن‌بست خواهد خورد، در حالی‌که متغیر ایستا توسط یک روش ایمن نخ مقداردهی اولیه می‌شود.

### ۲,۷,۲ راه‌حل سازگار

این راه‌حل، از مقداردهی اولیه‌ی آرایه‌ی محلی ایستای *cache* ممانعت به عمل می‌آورد. در عوض، به مقداردهی اولیه‌ی صفر، جهت تعیین آن‌که آیا هر عضو از آرایه هنوز به مقداری تنظیم شده‌است، و در صورت عدم تنظیم، مقدار آن به‌صورت بازگشتی محاسبه می‌شود، تکیه دارد. سپس، مقدار کَش‌شده را، زمانی‌که به مقدار محتمل یا محاسبه‌شده نیاز است، بازمی‌گرداند.

```
#include <stdexcept>
int fact(int i) noexcept(false)
{
	if (i < 0)
	{
		// Negative factorials are undefined.
		throw std::domain_error("i must be >= 0");
	}
	// Use the lazy-initialized 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;
}
```

### ۳,۷,۲ نمونه‌کد ناسازگار

در این نمونه‌کد، مقدار *numWheels* در file1.cpp، به مقداردهی اولیه‌ی *c*، وابسته است. اگرچه، به‌دلیل آن‌که *c* در یک واحد ترجمه‌ی متفاوت از *numWheels* (file2.cpp) تعریف می‌شود، هیچ تضمینی وجود ندارد که *c*، پیش از آن‌که numWheels با فراخوانی *c.get_num_wheels ()* مقداردهی اولیه گردد، با فراخوانی *get_default_car ()* مقداردهی اولیه شود. غالبا از این امر به‌عنوان "شکست مفتضحانه‌ی ترتیب مقداردهی اولیه‌ی ایستا" یاد می‌شود و یک رفتار نامشخص را نتیجه می‌دهد.

مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

```
// 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();
```

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

### ۱,۳,۷,۲ جزئیات پیاده‌سازی

غالبا، مقدار چاپ‌شده‌ی جریان خروجی استاندارد، بر ترتیب واحدهای ترجمه‌ی پیوسته و متصل تکیه دارد. به‌عنوان مثال، با اجرای دستور *clang++ file1.cpp file2.cpp && ./a.out* در Clang 3.8.0 روی لینوکس x86، مقدار صفر درج خواهد شد؛ در حالی‌که، دستور *clang++ file2.cpp file1.cpp && ./a.out*، مقدار ۶ را خواهد نوشت.

### ۴,۷,۲ راه‌حل سازگار

راه‌حل سازگار، از اصطلاح "ساخت در نخستین استفاده" جهت حل مشکل ترتیب مقداردهی اولیه‌ی ایستا بهره می‌جوید. کدهای متعلق به file.h و file2.cpp غیر قابل تغییر هستند. تنها، *numWheels* ایستای موجود در file1.cpp به بدنه‌ی یک تابع منتقل می‌شود. در نتیجه، تضمین می‌شود که مقداردهی اولیه‌ی *numWheels* زمانی روی دهد که کنترل، در نقطه‌ی اعلان (که کنترل را در طول ترتیب ضمانت می‌کند)

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

جریان یابد. شی عمومی *c*، پیش از آن‌که اجرای *main ()* آغاز شود، مقداردهی اولیه می‌گردد. بنابراین، زمانی‌که *get_num_wheels ()* فراخوانی می‌شود، تضمین می‌گردد که *c* قبلا به‌صورت پویا مقداردهی اولیه شده‌است.

```
// 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();
```

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه‌ای

### ۵.۷.۲ ارزیابی خطر

ورود مجدد و بازگشتی یک تابع طی مقداردهی اولیه‌ی یکی از اشیای ایستای آن، می‌تواند منجر شود تا حمله‌ی سقوط یا انکار خدمات رخ دهد. مقداردهی نخستین پویای مرتب نامعین می‌تواند رفتار نامشخص ناشی از دسترسی به یک شی مقداردهی اولیه‌نشده را در پی داشته باشد.

| قانون | شدت/دقت | احتمال | هزینه‌ی اصلاح | اولویت | سطح |
|---|---|---|---|---|---|
| DCL56-CPP | پایین | بعید | متوسط | P2 | L3 |

## ۸,۲ DCL57-CPP به استثنائات اجازه ندهید از توابع مخرب یا آزادساز فرار کنند

تحت شرایط خاص، خاتمه دادن به یک مخرب *operator delete* یا *operator delete []* از طریق ایجاد یک استثنا می‌تواند منجر به یک رفتار تعریف‌نشده شود. در چنین شرایطی، لازم است تابع به‌صورت *noexcept* اعلان شود، زیرا ممکن است ایجاد یک استثنا از تابع، رفتار خوبی تلقی نشود.

توابع آزادسازی (شی، آرایه، و جایابی فرم‌ها در حوزه‌های عمومی یا کلاس) نباید با ایجاد یک استثنا خاتمه یابند. چنین توابعی را به‌صورت *noexcept (false)* تعریف نکنید. اگرچه، قابل قبول است که بر مشخصه‌ی *noexcept (true)* ضمنی تکیه کند و یا *noexcept* را صریحا در امضای تابع اعلان نماید.

احتمالا، مخرب‌های شی طی فرآیند فراخوانی پشته، به‌عنوان حاصل استثنائی که در حال ایجاد شدن است، فراخوانی می‌شوند. اگر مخرب، خود، یک استثنا را ایجاد کند، به‌عنوان حاصل استثنائی که در حال ایجاد شدن است، فراخوانی شده‌است. سپس، تابع *std::terminate ()*، متأثر از فراخوانی پیش‌فرض *std::abort ()*، فراخوانی می‌شود. وقتی *std::abort ()* فراخوانی می‌گردد، اشیای بیشتری خراب نمی‌شوند. در نتیجه، شاهد برنامه و رفتاری نامشخص خواهیم بود. هرگز مخربی را از طریق ایجاد یک استثنا خاتمه ندهید.

یک اعلان ضمنی یک مخرب به‌گونه‌ای در نظر گرفته می‌شود تا *noexcept (true)* باشد. بنابراین، مخرب‌ها نباید *noexcept (false)* اعلان شوند. اما در عوض ممکن است بر *noexcept (true)* ضمنی دلالت کنند یا صریحا، *noexcept* اعلان شوند.

### ۱,۸,۲ نمونه‌کد ناسازگار

در این نمونه‌کد، مخربِ کلاس، *noexcept* ضمنی را تضمین نمی‌کند. زیرا، امکان دارد استثنائی را حتی به‌عنوان نتیجه‌ای از یک استثنای پرتاب‌شده، ایجاد نماید. متعاقبا، به‌عنوان *noexcept (false)* اعلان می‌شود، اما، هنوز می‌تواند منجر به رفتار نامشخصی گردد.

مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

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

### ۲.۸.۲ نمونه‌کد ناسازگار (*std::uncaught_exception ()*)

همان‌گونه که در این مثال مشاهده می‌شود، اگر استثنای در حال پردازشی وجود داشته‌باشد، استفاده از *std::uncaught_exception ()* در مخرب، مشکل خاتمه یافتن از طریق اجتناب از انتشار استثنا را برطرف می‌سازد. اگرچه، ممکن است این روش با دور زدن پردازش مخرب عادی، آن را از نشر منابع مهم حفظ نماید.

```
#include <exception>
#include <stdexcept>
class S
{
	bool has_error() const;
	public:
		~S() noexcept(false)
		{
			// Normal processing
			if (has_error() && !std::uncaught_exception())
			{
			throw std::logic_error("Something bad");
			}
		}
};
```

### ۳.۸.۲ نمونه‌کد ناسازگار (*function_try_block*)

این نمونه‌کد، و نیز راه‌حل سازگار پیرو آن، فرض می‌کنند که یک کلاس *Bad* حاوی مخربی وجود دارد، که می‌تواند عملیات ایجاد را انجام دهد. با وجود آن‌که کلاس این قانون را نقض می‌نماید، فرض می‌شود کلاس قادر نیست در راستای تفسیر قانون مذکور تغییر یابد.

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

```
// Assume that this class is provided by a 3rd party and it is not something that can be modified by the user.
class Bad
{
	~Bad() noexcept(false);
};
```

در نتیجه، استثنای گرفتار، به ناچار از بند مخرب *SomeClass* خواهد گریخت. زیرا، زمانی‌که کنترل به انتهای کنترل‌گر *function_try_*block می‌رسد، مجددا به‌صورت ضمنی ایجاد می‌شود.

```
class SomeClass
{
	Bad bad_member;
	public:
		~SomeClass() try
		{
			// ...
		}
		catch(...)
		{
			// Handle the exception thrown from the Bad destructor.
		}
};
```

### ۴,۸,۲ راه‌حل سازگار

یک مخرب باید مسیری را طی نماید که هیچ استثنای فعالی در آن وجود نداشته باشد. به‌طورکلی، مخرب باید تنها عملیاتی را، که استثنائات را ایجاد نمی‌کند، فراخوانی نماید؛ و یا لازم است تمامی استثنائات را مدیریت کند و آنها را، حتی به‌صورت ضمنی، مجددا ایجاد ننماید. این راه‌حل سازگار، به‌دلیل استفاده از یک عبارت *return* صریح در مخرب *SomeClass*، متفاوت از نمونه‌های ناسازگار پیشین است. این عبارت، از امکان کنترل انتهای کنترل‌گر استثنا، ممانعت به عمل می‌آورد. در نتیجه، زمانی‌که *bad*_member تخریب می‌شود، کنترل‌گر، استثنای مذکور ایجادشده را توسط *Bad::~Bad ()* خواهد گرفت. همچنین، هر استثنای ایجادشده‌ای را داخل عبارت ترکیبی *function_try_block* می‌گیرد، اما مخرب *SomeClass* را به‌وسیله‌ی ایجاد یک استثنا خاتمه نخواهد داد.

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

```
class SomeClass
{
        Bad bad_member;
        public:
                ~SomeClass() try
                {
                        // ...
                }
                catch(...)
                {
                        // Catch exceptions thrown from noncompliant destructors of  member objects or base
                        // class subobjects.
                        // NOTE: Flowing off the end of a destructor function-try-block causes the caught
                        // exception to be implicitly rethrown, but an explicit return statement will prevent that
                        // from happening.
                        return;
                }
};
```

### ۵,۸,۲ نمونه‌کد ناسازگار

در این نمونه‌کد، یک آزادسازی عمومی *noexcept (false)* اعلان می‌شود و اگر برخی شرایط به‌درستی اِعمال نشوند، استثنائی را ایجاد می‌کند. اگرچه، ایجاد استثنا توسط یک تابع آزادسازی، منجر به رفتاری نامشخص خواهد شد.

```
#include <stdexcept>
bool perform_dealloc(void *);
void operator delete(void *ptr) noexcept(false)
{
        if (perform_dealloc(ptr))
        {
                throw std::logic_error("Something bad");
        }
}
```

### ۶,۸,۲ راه‌حل سازگار

این راه‌حل، استثنائات را در حادثه‌ای، که آزادسازی شکست می‌خورد، ایجاد نمی‌کند. اما در عوض، احتمال وقوع شکست‌ها بسیار محتمل است.

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

```
#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); // Fail, but still call destructors
	}
}
```

### ۷.۸.۲ ارزیابی خطر

تلاش برای ایجاد استثنائات از توابع مخرب یا آزادساز می‌تواند رفتار نامشخصی را در پی داشته‌باشد و منجر به گم شدن منابع و یا حملات انکار خدمات شود.

| قانون | شدت/دقت | احتمال | هزینه‌ی اصلاح | اولویت | سطح |
|---|---|---|---|---|---|
| DCL57-CPP | پایین | احتمال زیاد | متوسط | P6 | L3 |

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

## ۹,۲ DCL58-CPP فضاهای استاندارد را تغییر ندهید

فضاهای نام[۲۰]، نواحی اعلان جدیدی را جهت اعلان‌ها، با کاهش احتمال تصادم شناسه‌ها با سایر نواحی اعلان، معرفی می‌کنند. یکی از ویژگی‌های فضای نام این است که می‌توانند، حتی در واحدهای ترجمه‌ی مجزا، توسعه یابند. به‌عنوان مثال، اعلان زیر، اعلانی خوش‌ترکیب است.

```
namespace MyNamespace
{
        int i;
}
namespace MyNamespace
{
        int i;
}
void f()
{
        MyNamespace::i = MyNamespace::i = 12;
}
```

کتابخانه‌ی استاندارد، فضای نام *std* را برای اعلان‌های استانداردمحور، مانند *std::string.std::vector* و *std::for_each*، معرفی می‌کند. اگرچه، معرفی اعلانات جدید در فضای نام *std* و تحت شرایط خاص، یک رفتار نامشخص است.

اعلانات یا تعاریف جدید را به فضای نام *std* یا *posix*، و یا به یک فضای نام حاوی آن اضافه نکنید؛ مگر برای یک الگوی تخصصی، که به نوع تعریف‌شده توسط کاربر نیاز دارد تا نیازمندی‌های کتابخانه‌ای استاندارد الگوهای اصلی را برطرف سازد.

گروه کاری کتابخانه‌ای، مسئول تبیین بخش کتابخانه‌ی استاندارد C++ هستند. این گروه، با معضلی در تعریف نوع تعریف‌شده توسط کاربر مواجه هستند. اگرچه، گروه کاری کتابخانه‌ای هیچ موضع رسمی بر تعریف ندارد، ما آن را به‌گونه‌ای تعریف می‌کنیم که هر *class.struct.union* یا *enum* در فضای نام *std* و یا فضای نام حاوی *std*، تعریف نشده باشد. به‌طور مؤثر، این حالت به جای آن‌که مبتنی بر کتابخانه‌ی استاندارد باشد، کاربرمحور است.

### ۱,۹,۲ نمونه‌کد ناسازگار

در این نمونه‌کد، اعلان *x* به فضای نام *std* اضافه می‌شود. این امر منجر به یک رفتار نامشخص خواهد شد.

---

[۲۰] Namespaces

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه‌ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

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

### ۲.۹.۲ راه‌حل سازگار

در این راه‌حل، فرض بر آن است که برنامه‌نویس قصد دارد جهت جلوگیری از تصادم با سایر شناسه‌های عمومی، اعلان *x* را در یک فضای نام قرار دهد. از این‌روی، به جای آن‌که اعلان در یک فضای نام *std* قرار داده شود، در فضای نامی که فاقد اسم ذخیره‌شده است جای می‌گیرد.

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

### ۳.۹.۲ نمونه‌کد ناسازگار

در این نمونه‌کد، یک الگوی تخصصی از *std::plus* به فضای نام *std* اضافه می‌شود تا به *std::plus* اجازه دهد یک *std::string* و شی *MyString* را الحاق کند. اگرچه، به‌دلیل آن‌که الگوی تخصصی، از نوع مبتنی بر کتابخانه‌ی استاندارد است (*std::string*)، این کد یک رفتار نامشخص را نتیجه خواهد داد.

```
#include <functional>
#include <iostream>
#include <string>
class MyString
{
	std::string data;
	public:
		MyString(const std::string &data) : data(data) {}
		const std::string &get_data() const
		{
			return data;
		}
};
namespace std
{
	template <> struct plus<string> : binary_function<string, MyString, string>
	{
		string operator()(const string &lhs, const MyString &rhs) const
		{
			return lhs + rhs.get_data();
		}
	};
}
```

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

```
void f()
{
	std::string s1("My String");
	MyString s2(" + Your String");
	std::plus<std::string> p;
	std::cout << p(s1, s2) << std::endl;
}
```

### ۴,۹,۲ راه‌حل سازگار

واسط *std::plus* نیاز دارد که هر دو پارامتر عملگر فراخوان تابع و نیز نوع بازگشتی، هم‌نوع باشند. به‌دلیل آن‌که تخصص به کاررفته در نمونه‌کد ناسازگار منجر به رفتار تعریف‌نشده‌ای می‌شود، راه‌حل سازگار مورد مطالعه، یک *std::binary_function* جدید اشتقاقی را، که می‌تواند یک *std::string* را بدون نیاز به تغییر نام فضای *std*، به یک شی *MyString* اضافه نماید، تعریف می‌کند.

```
#include <functional>
#include <iostream>
#include <string>
class MyString
{
	std::string data;
	public:
		MyString(const std::string &data) : data(data) {}
		const std::string &get_data() const
		{
			return data;
		}
};
struct my_plus
	: std::binary_function<std::string, MyString, std::string>
	{
		std::string operator()(const std::string &lhs, const MyString &rhs) const
		{
			return lhs + rhs.get_data();
		}
	};
void f()
{
	std::string s1("My String");
	MyString s2(" + Your String");
	my_plus p; std::cout << p(s1, s2) << std::endl;
}
```

### ۵,۹,۲ راه‌حل سازگار

در این راه‌حل، تخصصی از *std::plus* به فضای نام *std* اضافه می‌شود. اما این تخصص، مبتنی بر نوع کاربر است و نیازمندی‌های کتابخانه‌ای الگوی استاندارد را برای الگوهای اصلی برطرف می‌سازد. اگرچه، به‌دلیل

مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

آن‌که *MyString* می‌تواند از *std::string* ساخته شود، این راه‌حل شامل فراخوانی یک سازنده‌ی مبدل است، که در راه‌حل سازگار پیشین تعبیه نشده‌بود.

```
#include <functional>
#include <iostream>
#include <string>
class MyString
{
	std::string data;
	public:
		MyString(const std::string &data) : data(data) {}
		const std::string &get_data() const
		{
			return data;
		}
};
namespace std
{
	template <>
	struct plus <MyString>
	{
		MyString operator()(const MyString &lhs, const MyString &rhs) const
		{
			return lhs.get_data() + rhs.get_data();
		}
	};
}
void f()
{
	std::string s1("My String");
	MyString s2(" + Your String");
	std::plus<MyString> p;
	std::cout << p(s1, s2).get_data() << std::endl;
}
```

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه

### ۶,۹,۲ ارزیابی خطر

تغییر فضای نام استاندارد می‌تواند منجر به رفتار نامعین در کتابخانه‌ی استاندارد C++ شود.

| قانون | شدت/دقت | احتمال | هزینه‌ی اصلاح | اولویت | سطح |
|---|---|---|---|---|---|
| **DCL58-CPP** | بالا | بعید | متوسط | P6 | L2 |

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

## ۱۰,۲ DCL59-CPP یک فضای بدون نام را در یک فایل سرآیند تعریف نکنید

فضاهای نام فاقد اسم جهت تعریف فضای نامی استفاده می‌شوند که برای واحد ترجمه، منحصر به‌فرد است و نام‌های مشمول در آن حاوی پیوند داخلی پیشوند هستند.

غالبا، کد C++، از کیفیت تولید فایل‌های سرآیند به‌عنوان ابزاری جهت اشتراک کد میان واحدهای ترجمه استفاده می‌کند. یک فایل سرآیند، فایلی است که از طریق دستور *#include* در یک واحد ترجمه قرار می‌گیرد. یک فضای نام فاقد اسم را در یک فایل سرآیند تعریف نکنید. زیرا، این امر نتایج غیرمترقبه‌ای را به‌همراه خواهد داشت. باتوجه به پیوند داخلی پیش‌فرض، هر واحد ترجمه، نمونه‌ی منحصر به‌فرد شخصی خود را از اعضای فضای نام فاقد اسمی که در واحد ترجمه، مبتنی بر قانون تک‌توصیفی[۲۱] (ODR) هستند، تعریف خواهد نمود. این امر می‌تواند نتایج غیرمترقبه‌ای، از قبیل تورم نتایج قابل اجرا و یا رفتار نامعین غیرعمد ناشی از نقض ODR را با خود به‌همراه داشته باشد.

### ۱,۱۰,۲ نمونه‌کد ناسازگار

در این نمونه‌کد، متغیر *v* در فضای نام فاقد اسم در فایل سرآیند، تعریف می‌شود و توسط دو واحد ترجمه قابل دست‌یابی است. هر واحد ترجمه، مقدار کنونی *v* را چاپ می‌کند و سپس، یک مقدار جدید به آن اختصاص می‌دهد. اگرچه، به‌دلیل آن‌که *v* در یک فضای نام فاقد اسم تعریف می‌شود، هر واحد ترجمه بر نمونه‌ی شخصی خود از *v* عمل می‌کند، که منجر به تولید یک خروجی غیرمنتظره خواهد شد، که در ادامه مشاهده می‌گردد.

```
// a.h
#ifndef A_HEADER_FILE
#define A_HEADER_FILE
namespace
{
        int v;
}
#endif // A_HEADER_FILE
// a.cpp
#include "a.h"
#include <iostream>
```

[۲۱] One Definition Rule (ODR)

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

```
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();
}
```

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

### ۲.۱۰.۲ راه‌حل سازگار

در این راه‌حل، $v$ تنها در یک واحد ترجمه تعریف می‌شود. اما، به‌صورت خارجی برای تمامی واحدهای ترجمه قابل مشاهده‌است. در نتیجه، شاهد رفتار مورد انتظاری خواهیم بود.

```
// 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;
        // ...
}
```

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه‌ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

```
// 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
 }
```

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

### ۳.۱۰.۲ نمونه‌کد ناسازگار

در این نمونه‌کد، متغیر *v* در یک فضای نام فاقد اسم در فایل سرآیند تعریف می‌شود. همچنین، یک تابع درون‌خطی *get_v ()* تعریف می‌گردد، که به متغیر *v* دسترسی دارد. قانون تک‌توصیفی، که از تابع درون‌خطی چندین واحد ترجمه استفاده می‌نماید (همان‌گونه که در اجرای *f ()* و *g ()* نمایان است)، ODR را نقض می‌کند. زیرا، تعریف *get_v ()* در تمامی واحدهای ترجمه، یکسان نیست. چراکه هر واحد ترجمه، به یک *v* منحصر به‌فرد ارجاع می‌دهد.

```
// a.h
#ifndef A_HEADER_FILE
#define A_HEADER_FILE
namespace
{
	int v;
}
inline int get_v()
{
	return v;
}
#endif // A_HEADER_FILE
// a.cpp
#include "a.h"
```

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

```
void f()
{
        int i = get_v();
        // ...
}
// b.cpp
#include "a.h"
void g()
{
        int i = get_v();
        // ...
}
```

### ۴,۱۰,۲ راه‌حل سازگار

در این راه‌حل، *v* تنهادر یک واحد ترجمه می‌شود. اما، به‌صورت خارجی برای تمامی واحدهای ترجمه، قابل مشاهده و از طریق تابع *get_v ()* قابل دست‌یابی است.

```
// a.h
#ifndef A_HEADER_FILE
#define A_HEADER_FILE
Extern int v;
inline int get_v()
{
        return v;
}
#endif // A_HEADER_FILE
// a.cpp
#include "a.h"
// Externally used by get_v();
int v;
void f()
{
        int i = get_v();
        // ...
}
// b.cpp
#include "a.h"
void g()
{
        int i = get_v();
        // ...
}
```

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه‌ای

### ۵,۱۰,۲نمونه‌کد ناسازگار

در این نمونه‌کد، تابع *f()*، در یک فایل سرآیند تعریف می‌شود. اگرچه، فراخوانی فایل سرآیند در چندین واحد ترجمه، منجر به نقضی از ODR می‌شود، که معمولا به‌دلیل وجود تعاریف متعدد از یک تابع با همان نام، تشخیص خطای تولیدشده در زمان پیوند را به‌همراه خواهد داشت.

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

### ۶,۱۰,۲نمونه‌کد ناسازگار

این نمونه‌کد سعی دارد که خطاهای زمانی پیوند را با تعریف *f()* در یک فضای نام فاقد اسم، برطرف سازد. اگرچه، چندین تابع منحصر به‌فرد از *f()* تولید می‌کند، که قابل اجرا هستند. اگر یک *a.h* از واحدهای ترجمه‌ی متعدد ناشی شود، می‌تواند منجر به افزایش زمان پیوند، یک فایل اجرایی بزرگتر، و نیز کاهش کارایی شود.

```
// 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"
// ...
```

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

### ۷.۱۰.۲ راه‌حل سازگار

در این راه‌حل، *f()* در یک فضای نام فاقد اسم تعریف نمی‌شود، بلکه، به‌عنوان یک تابع درون‌خطی تعریف می‌گردد. توابع درون‌خطی باید به‌صورت یکسان و در تمامی واحدهای ترجمه‌ای، که در آنها استفاده می‌شوند، تعریف گردند. این امر به یک پیاده‌سازی اجازه می‌دهد تا نمونه‌ای منحصر به‌فرد از تابع را در زمان اجرای رخداد اصلی تابع، که برای هیچ منبع فراوانی ایجاد نمی‌شود، تولید کند.

```
// 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"
// ...
```

### ۸.۱۰.۲ ارزیابی خطر

تعریف یک فضای نام فاقد اسم در یک فایل سرآیند می‌تواند منجر به نقض جامعیت داده‌ها و مشکلات کارایی و اجرایی شود. اما، بعید است که از تست‌های کافی به دور بماند. نقض ODR منجر به رفتار تعریف‌شده خواهد شد.

| قانون | شدت/دقت | احتمال | هزینه‌ی اصلاح | اولویت | سطح |
|---|---|---|---|---|---|
| **DCL59-CPP** | متوسط | بعید | متوسط | P4 | L3 |

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

## ۱۱,۲ DCL60-CPP از قانوت تک‌توصیفی پیروی کنید

عموما برنامه‌های C++ کوچک اما بااهمیت، به چندین واحد ترجمه تقسیم می‌شوند، که بعدا به یکدیگر می‌پیوندند تا نمونه‌ای اجرایی را تشکیل دهند. زبان C++ برای پوشش چنین مدلی، تعاریف شی نام‌دار را محدود می‌سازد تا تضمین کند که پیوند صرفا با ابراز نیاز به یک تک‌تعریف برای یک شی در واحدهای ترجمه، رفتار خواهد کرد. این مدل، ODR نامیده می‌شود.

رایج‌ترین روش جهت تفسیر چندین واحد ترجمه، شامل اعلاناتی است که در یک فایل سرآیند مقیم هستند و متعاقبا، از طریق *#include* برای فایل مرجع قابل دسترس خواهد بود. غالبا این اعلانات، تعاریف نیز به‌شمار می‌روند، مانند تعاریف کلاس و الگوی تابع.

تعریفی که در دو واحد ترجمه‌ی مجزا، توسط یک دستور *#include* معرفی می‌شود، ODR را نقض نخواهد کرد. زیرا، توصیفات در هر دو واحد ترجمه یکسان هستند.

اگرچه، ممکن است مواردی از قبیل ODR یک تعریف که توسط *#include* و با استفاده از مشخصه‌های پیوند زبان بلاک معرفی‌شده، توسعه‌های زمانی خاص فروشنده، و مواردی مشابه نقض شوند. یک سناریوی محتمل برای نقض ODR این است که تعاریف تصادفی اشیای متنوع در واحدهای ترجمه‌ی گوناگون وجود داشته باشند. بنابراین، قانون تک‌توصیفی را نقض نکنید. زیرا، رفتار تعریف‌نشده‌ای را با خود به‌همراه خواهد داشت.

### ۱,۱۱,۲ نمونه‌کد ناسازگار

در این نمونه‌کد، دو واحد ترجمه‌ی متنوع، یک کلاس را با نام مشابه، اما با تعاریف متنوع، تعریف خواهند کرد. اگرچه، دو تعریف مذکور به‌صورت کاربردی، یکسان هستند (هر دو، کلاسی با نام *S* را توسط یک عضو داده‌ای تک، که عمومی و غیرایستا است و *int a* نام دارد، تعریف می‌کنند)، با استفاده از ترتیبی یکسان از نشانه‌ها تعریف نمی‌شوند. نمونه‌کد مذکور، ODR را نقض می‌کند و رفتار تعریف‌نشده‌ای از خود بروز می‌دهد.

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

```
// a.cpp
struct S
{
        int a;
};
 // b.cpp
class S
{
        public:
                int a;
};
```

### ۲.۱۱.۲ راه‌حل سازگار

کاهش صحیح، به هدف برنامه‌نویس بستگی دارد. اگر برنامه‌نویس قصد داشته باشد که جهت استفاده‌ی مشترک، همان تعریف کلاس در هر دو واحد ترجمه قابل رؤیت باشند، راه‌حل بهینه، استفاده از یک فایل سرآیند برای معرفی شی در هر دو واحد ترجمه است.

```
// S.h
struct S
{
        int a;
};
// a.cpp
#include "S.h"
// b.cpp
#include "S.h"
```

### ۳.۱۱.۲ راه‌حل سازگار

اگر نقض ODR ناشی از تصادم اسم تصادفی باشد، بهترین راه‌حل تسکینی، تضمین این است که تعاریف کلاس‌ها منحصر به‌فرد باشند. متناوبا، کلاس‌ها می‌توانند اسامی متمایزی را در هر واحد ترجمه اتخاذ کنند تا از نقض ODR جلوگیری به عمل آید.

```
// a.cpp
namespace
{
        struct S
        {
                int a;
        };
}
```

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

```
// b.cpp
namespace
{
        class S
        {
                public: int a;
         };
}
```

### ۴,۱۱,۲ نمونه‌کد ناسازگار (مایکروسافت ویژوال استودیو)

در این نمونه‌کد، یک تعریف از کلاس، با استفاده از *#include* به دو واحد ترجمه معرفی می‌شود. اگرچه، یکی از واحدهای ترجمه، از *#pragma* اجرا-محوری استفاده می‌کند که مایکروسافت ویژوال استودیو آن را برای تعیین نیازمندی‌های تخصیص فیلد ساختاری پوشش می‌دهد. متعاقبا، شاید دو کلاس تعریف، شامل طرح‌های متفاوتی از هر واحد ترجمه باشند، که نقض ODR را به‌همراه خواهد داشت.

```
// s.h
struct S
{
        char c;
        int a;
};
void init_s(S &s);
// s.cpp
#include "s.h"
void init_s(S &s);
{
        s.c = 'a';
        s.a = 12;
} //
a.cpp
#pragma pack(push, 1)
#include "s.h"
#pragma pack(pop)
void f()
{
        S s;
        init_s(s);
}
```

*۱,۴,۱۱,۲ جزئیات پیاده‌سازی*

ممکن است نمونه‌کد ناسازگار پیشین، فضایی را به یک شی در *a*.cpp تخصیص دهد که با اندازه‌ی مورد نظر *int_s ()* در *S.cpp* تفاوت داشته باشد. وقتی *S.cpp* ترجمه می‌شود، ممکن است طرح ساختاری، شامل بایت‌های پدینگ میان اعضای داده‌ای *c* و *a* شود. زمانی‌که *z.cpp* ترجمه می‌شود، این امکان وجود دارد که طرح ساختاری آن، بایت‌های پدینگ را به‌عنوان نتیجه‌ای از دستور *#pragma pack* حذف کند. بنابراین،

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

ممکن است شی گذرا به *int_s ()* کوچکتر از آنه انتظار می‌رود باشد. در نتیجه، وقتی *int_s ()*، اعضای داده‌ای *S* را مقداردهی اولیه می‌کند، این احتمال وجود دارد که سربار بافر اتفاق بیفتد.

### ۵,۱۱,۲راه‌حل سازگار

در این راه‌حل، دستور تخصیص عضو ساختار مبتنی بر اجرا حذف شده، تا تضمین نماید که تمامی تعاریف *S* توسط ODR تفسیر می‌شوند.

```
struct S
{
        char c;
        int a;
};
void init_s(S &s);
// s.cpp
#include "s.h"
void init_s(S &s);
{
        s.c = 'a';
        s.a = 12;
}
// a.cpp
#include "s.h"
void f()
{
        S s;
        init_s(s);
}
```

### ۶,۱۱,۲نمونه‌کد ناسازگار

در این نمونه‌کد، شی ثابت *n* دارای پیوند داخلی است. اما، این شی در بدنه‌ی *f ()* مورد استفاده‌ی ODR است و پیوند خارجی دارد. به‌دلیل آن‌که *f ()* به‌عنوان یک تابع درون‌خطی اعلان می‌شود، باید تعریف *f ()* در تمامی واحدهای ترجمه، یکسان باشد. اگرچه، هر واحد ترجمه دارای نمونه‌ای منحصر به‌فرد از *n* است، که منجر به نقض ODR می‌شود.

```
const int n = 42;
int g(const int &lhs, const int &rhs);
inline int f(int k)
{
        return g(k, n);
}
```

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

### ۷.۱۱.۲ راه‌حل سازگار

این راه‌حل باید یکی از سه فاکتور زیر را تغییر دهد:

- نباید در *f()* ، *n* را در راستای ODR به کار گیرد.
- باید *n* را به‌گونه‌ای اعلان کند که پیوند خارجی داشته باشد.
- نباید از تعریف درون‌خطی *f()* استفاده کند.

اگر شرایط اجازه دهد تا برای تغییر امضای *g()*، پارامترها با مقادیر مقدار مرجع، جایگزین شوند، آنگاه *n* در *f()*، مورد استفاده‌ی ODR نخواهد بود. زیرا، *n* به‌عنوان یک عبارت ثابت تعیین می‌شود. این راه‌حل، سازگار است اما منطقی نیست. شاید تغییر امضای *g()*، مطلوب نباشد؛ به‌عنوان مثال، در حالتی‌که *g()*، *std::max* را از طریق *<algorithm>* نشان دهد. همچنین، به‌دلیل پیوند متفاوت استفاده‌شده توسط *n* و *f()*، اگر تعریف *f()* به *n* مستفید از ODR تغییر یابد، هنوز احتمال نقوض تصادفی ODR وجود دارد.

```
const int n = 42;
int g(int lhs, int rhs);
inline int f(int k)
{
        return g(k, n);
}
```

### ۸.۱۱.۲ راه‌حل سازگار

در این راه‌حل، شی ثابت *n* با یک شمارنده‌ی هم‌نام جایگزین می‌شود. شمارش‌های نام‌گذاری‌شده‌ی تعریف‌شده در حوزه‌ی فضای نام، دارای همان پیوندی هستند به‌عنوان فضای نام، آن را دربر می‌گیرند. فضای نام عمومی، دارای پیوند خارجی است. بنابراین، توصیف شمارش نام‌دار و نیز شمارنده‌های آن، حاوی پیوند خارجی هستند. اگرچه، با مطلوبتی پایین‌تر، بارهای ناشی از نگهداری کدهای پیشین را تحمل نمی‌کند. زیرا، *n* و *f()* پیوند یکسانی دارند.

```
enum Constants
{
        N = 42
};
int g(const int &lhs, const int &rhs);
inline int f(int k)
{
        return g(k, N);
}
```

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

### ۹.۱۱.۲ ارزیابی خطر

نقض ODR منجر به رفتارهای نامعینی می‌شود که سوءاستفاده‌هایی را با عنوان حملات انکار خدمات با خود به‌همراه خواهد داشت. عدم اجرای ODR، حمله‌ی اشاره‌گر تابع مجازی ملقب به سوءاستفاده‌ی VPTR را ممکن می‌سازد. در این سوءاستفاده، یک شی از جدول تابع مجازی[۲۲] (vtable)، به‌گونه‌ای خراب می‌شود که یک تابع مجازی روی شی، منجر به کدهای مخرب قابل اجرا شود. شایان ذکر است، مهاجم باید جهت معرفی کلاس مخرب، به ساختمان کد سیستم دسترسی داشته باشد.

| قانون | شدت/دقت | احتمال | هزینه‌ی اصلاح | اولویت | سطح |
|---|---|---|---|---|---|
| DCL60-CPP | بالا | بعید | بالا | P3 | L3 |

مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

[۲۲] Virtual table (vtable)

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

# فصل سوم

# اصطلاحات (EXP)

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

## ۱,۳ EXP50-CPP به ترتیب ارزیابی برای تأثیرات جانبی وابسته نباشید

در C++، تغییر و اصلاح یک شی، فراخوانی یک تابع ورودی/خروجی کتابخانه‌ای، دسترسی به مقدار کیفی volatile یا فراخوانی تابعی که یکی از این فرمان‌ها را اجرا می‌کند، راه‌هایی برای تغییر و اصلاح وضعیت محیط اجرا به‌شمار می‌روند. این‌گونه فرمان‌ها، تاثیرات جانبی نامیده می‌شوند. تمام روابط موجود میان محسابه‌ی مقدار و تاثیرات جانبی می‌توانند به لحاظ توالی ارزیابی آنها توضیح داده شوند.

حالات زیر، دارای محدودیت‌های توالی بوده، که خارج از مرتب‌سازی مرحله‌ای معمول هستند:

- در عبارت‌های پسوندی "++" و "--"، محاسبه‌ی مقدار، پیش از تغییر و اصلاح عملوند انجام می‌پذیرد.
- در عبارت‌های منطقی "&&"، اگر دومین عبارت ارزیابی شود، هر محاسبه‌ی مقدار و تاثیر جانبی در ارتباط با عبارت اول، پیش از محاسبه‌ی مقدار و تاثیر جانبی در ارتباط با عبارت دوم انجام می‌گیرد.
- در عبارت‌های منطقی "||"، اگر دومین عبارت ارزیابی شود، هر محاسبه‌ی مقدار و تاثیر جانبی در ارتباط با عبارت اول، پیش از محاسبه‌ی مقدار و تاثیر جانبی در ارتباط با عبارت دوم انجام می‌پذیرد.
- در عبارت‌های شرطی "?:"، هر محاسبه‌ی مقدار و تاثیر جانبی در ارتباط با عبارت اول، پیش از محاسبه مقدار و تاثیر جانبی در ارتباط با عبارت دوم یا سوم انجام می‌گیرد (هرکدام که ارزیابی شود).
- در عبارت‌های تخصیصی (شامل عبارت‌های تخصیصی ترکیبی)، تخصیص پس از محاسبات مقدار عملوندهای چپ و راست و پیش از محاسبه‌ی مقدار عبارت تخصیصی انجام می‌پذیرد.
- در عبارت‌های کاما ","، هر محاسبه‌ی مقدار و تاثیر جانبی مرتبط با عبارت سمت چپ، پیش از محاسبه‌ی مقدار و تاثیر جانبی مرتبط با عبارت سمت راست انجام می‌شود.
- هنگام ارزیابی لیست‌های مقداردهندگان اولیه، محاسبه‌ی مقدار و تاثیر جانبی مرتبط با هر عبارت مقداردهنده‌ی اولیه، پیش از هر محاسبه‌ی مقدار و تاثیر جانبی مرتبط با عبارت مقداردهنده‌ی بعدی انجام می‌پذیرد.
- هنگامی‌که یک کنترل‌کننده‌ی سیگنال به‌عنوان نتیجه‌ی فراخوانی برای *std : : raise()* اجرا می‌شود، اجرای آن بعد از احضار *std : : raise()* و قبل از بازگشت انجام می‌گیرد.

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

- تکامل مخرب‌های اشیای مقداردهی اولیه‌شده با دوره‌ی زمانی ذخیره‌سازی نخ، داخل یک نخ، پیش از شروع تخریب هر شی با دوره‌ی زمانی ذخیره‌سازی ایستا انجام می‌شود.
- در یک عبارت جدید، مقداردهی اولیه‌ی یک شی اختصاص داده‌شده، پیش از محاسبه‌ی ارزش آن انجام می‌پذیرد.
- هنگامی‌که یک کلاس سازنده‌ی پیش‌فرض برای مقداردهی اولیه‌ی یک عنصر از یک آرایه، فراخوانی می‌شود و این کلاس سازنده، حداقل دارای یک پارامتر پیش‌فرض است، تخریب هر پارامتر موقت ساخته‌شده در پارامتر پیش‌فرض، پیش از تخریب عنصر بعدی آرایه انجام می‌شود.
- تخریب پارامترهای موقتی، که دوره‌ی زمانی آنها توسط مرجع تمدید نشده‌است، پیش از تخریب پارامترهای موقتی است که پیشتر، در همان عبارت کامل، ساخته می‌شوند.
- توابع مرتب‌سازی حافظه‌ی پویا می‌توانند به‌صراحت، مرتب‌سازی مرحله‌ای را برای عبارات تعیین کنند.

این قانون بدان معنا است که وضعیت‌هایی، از قبیل آنچه که در ادامه مشاهده می‌شود

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

دارای رفتار تعریف‌شده‌ای هستند و وضعیت‌هایی مانند موارد زیر را انجام نمی‌دهند:

```
// i is modified twice in the same full expression
i = ++i + 1;
// i is read other than to determine the value to be stored
a[i++] = i;
```

تمامی مثال‌های کاما در کد C++، به‌معنای استفاده از عملگر کاما نیست. به‌عنوان مثال، کامای موجود میان پارامترها در یک فراخوانی تابع، معادل عملگر کاما نیست. همچنین، عملگرهای سربار، حاوی عملوندهایی هستند که برای عملگر به مثابه پارامتر فراخوانی تابع عمل می‌کنند. از این‌روی، عملگرهای سربار مذکور، همانند فراخوانی توابع رفتار می‌نمایند.

### ۱.۱.۳ نمونه‌کد ناسازگار

در این نمونه‌کد، $i$ بیش از یک‌بار در روش غیرمرحله‌ای ارزیابی می‌شود. بنابراین، رفتار عبارت (اصطلاح)، تعریف‌نشده است.

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

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

### ۲.۱.۳ راه‌حل سازگار

این مثال‌ها، مستقل از ترتیب ارزیابی عملوندها هستند و هریک می‌تواند تنها به یک طریق تفسیر شود.

```
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];
        ++i;
        // ...
}
```

### ۳.۱.۳ نمونه‌کد ناسازگار

در این نمونه‌کد، فراخوانی *func ()* منجر به یک رفتار تعریف‌نشده می‌شود. زیرا اصطلاحات پارامتری، فاقد ترتیب هستند.

```
extern void func(int i, int j);
void f(int i)
{
        func(i++, i);
}
```

اصطلاح پارامتری نخست (چپ)، مقدار *i* را می‌خواند (تا مقداری را که باید ذخیره شود، مشخص کند) و سپس، آن را تعریف می‌نماید. پارامتر دوم (راست) مقدار *i* را می‌خواند، اما مشخص نمی‌کند که این مقدار باید در *i* ذخیره شود. این تلاش مجدد برای خواندن مقدار *i*، بیانگر همان رفتار تعریف‌نشده است.

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

### ۴,۱,۳ راه‌حل سازگار

این راه‌حل، زمانی مناسب است که برنامه‌نویس قصد دارد هر دو پارامتر *func ()* یکسان باشند.

```
extern void func(int i, int j);
void f(int i)
{
        i++;
        func(i, i);
}
```

راه‌حل زیر، زمانی مناسب است که برنامه‌نویس، پارامتر دوم را یک واحد بزرگتر از پارامتر اول در نظر بگیرد.

```
extern void func(int i, int j);
void f(int i)
{
        int j = i++;
        func(j, i);
}
```

### ۵,۱,۳ نمونه‌کد ناسازگار

این نمونه‌کد، مشابه نمونه‌ی قبل است. اگرچه، به جای فراخوانی مستقیم یک تابع ، یک *operator<<()* سربار را فراخوانی می‌کند. عملگرهای سربار، معادل توابع فراخوانی هستند و همان محدودیت‌های توابع فراخوانی را در مرحله‌بندی پارامترها شامل می‌شوند. در واقع، عملوندها از چپ به راست ارزیابی نمی‌شوند، بلکه با توجه به یکدیگر مورد ارزیابی قرار می‌گیرند. در نتیجه، این کد ناسازگار نیز منجر به رفتار تعریف نشده‌ای می‌شود.

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

### ۶,۱,۳ راه‌حل سازگار

دراین راه‌حل، دو فراخوانی برای *operator<<()* ایجاد شده‌است، که تضمین می‌کند پارامترها با ترتیب درستی چاپ شده‌اند.

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

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

### ۷.۱.۳ نمونه‌کد ناسازگار

ترتیب ارزیابی پارامترهای تابع، نامشخص است. این نمونه‌کد، بیانگر رفتار نامشخص است، نه رفتار تعریف‌نشده.

```
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());
}
```

ترتیب فراخوانی *a ()* و *b ()* نامشخص است. تنها، تضمین می‌شود که فراخوانی *a ()* و *b ()* پیش از فراخوانی *c()* انجام می‌گیرد. همان‌گونه که در این مثال مشاهده می‌شود، اگر *a ()* و *b ()* به یک وضعیت مشترک وابسته باشند، ممکن است هنگام محاسبه‌ی مقدار بازگشتی آنها، پارامترهای حاصل گذرا به *c ()*، میان مفسرها یا معماری‌ها متفاوت باشند.

### ۸.۱.۳ راه‌حل سازگار

در این راه‌حل، ترتیب ارزیابی *a ()* و *b ()* ثابت است. بنابراین، هیچگونه رفتار نامشخصی رخ نمی‌دهد.

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

```
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);
}
```

### ۹.۱.۳ ارزیابی خطر

ممکن است تلاش برای تغییر یک شی در ارزیابی نامرتب یا به‌صورت مرتب مشخص‌نشده، منجر شود که شی، مقدار غیرمنتظره‌ای را اتخاذ کند. در نهایت، برنامه رفتار غیرقابل انتظاری را از خود بروز خواهد داد.

| قانون | شدت/دقت | احتمال | هزینه‌ی اصلاح | اولویت | سطح |
|---|---|---|---|---|---|
| EXP50-CPP | متوسط | محتمل | متوسط | P8 | L2 |

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

## ۲٫۳ EXP51-CPP یک آرایه را توسط یک اشاره‌گر نادرست حذف نکنید

یک شی از یک آرایه را توسط اشاره‌گر ایستایی، که با نوع پویای اشاره‌گر به شی متفاوت است، حذف نکنید. حذف یک آرایه توسط یک اشاره‌گر نادرست، منجر به رفتار نامشخص می‌شود.

### ۱٫۲٫۳ نمونه‌کد ناسازگار

در این نمونه‌کد، آرایه‌ای از اشیای اشتقاقی ساخته شده و اشاره‌گر در *Base* * ذخیره شده‌است. علی‌رغم این‌که *Base::~Base ()* به‌صورت مجازی معرفی و شناسایی شده‌است، اما همچنان رفتار نامشخصی را به‌همراه خواهد داشت. علاوه بر این، تلاش برای اجرای اشاره‌گر محاسباتی روی نوع ایستای *Base* *، قانونی را، که تأکید دارد از اشاره‌گر محاسباتی روی اشیای چندریختی استفاده نشود، نقض می‌کند.

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

### ۲٫۲٫۳ راه‌حل سازگار

در این راه‌حل، نوع ایستای *b* از ساختار *Derived* * وجود دارد که رفتار نامشخص را هنگام نشانه‌گذاری به آرایه، حذف می‌کند.

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

تدوین مرکز امداد و هماهنگی عملیات رخدادهای رایانه‌ای

### ۳.۲.۳ ارزیابی خطر

تلاش برای ازبین بردن آرایه‌ای از اشیای چندریختی با استفاده از نوع ایستای نادرست، رفتار نامشخصی است که در عمل، منجر به اجرای نادرست برنامه و نشت اطلاعات در حافظه می‌شود.

| قانون | شدت/دقت | احتمال | هزینه‌ی اصلاح | اولویت | سطح |
|---|---|---|---|---|---|
| EXP51-CPP | پایین | بعید | متوسط | P2 | L3 |

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

## ۳,۳ EXP52-CPP در عملوندهای ارزیابی نشده، به تاثیرات جانبی وابسته نباشید

برخی اصطلاحات، شامل عملوندهایی ارزیابی‌نشده هستند. اصطلاحاتی که در ادامه مشاهده می‌کنید، عملوندهای خود را ارزیابی نمی‌کنند : *delval()*، *decltype()*، *noexcept()*، *typeid()*، و *sizeof()*.

به‌دلیل آن‌که این عملوندها در عبارات ارزیابی نمی‌شوند، هیچگونه تاثیر جانبی نیز در پی نخواهند داشت. اعتماد به این تاثیرات جانبی، منجر به رفتار دور از انتظاری خواهد شد. بنابراین، نباید به اینگونه تاثیرات جانبی ناشی از عملوندهای ارزیابی‌نشده وابسته بود.

زمانی عملوندهای ارزیابی‌نشده استفاده می‌شوند که نیاز است یک شی اعلان گردد، اما نیازی به تعریف آن شی نیست. به‌عنوان نمونه، در مثال زیر، از تابع *f ()*، که یک تابع سربار است، با تکیه بر عملوند ارزیابی‌نشده، جهت انتخاب سرباری بیش از حد انتظار و نیز برای تعیین نتیجه‌ی عبارت *sizeof ()* استفاده می‌شود.

```
int f(int);
double f(double);
size_t size = sizeof(f(0));
```

توابع و مثال‌هایی که از این الگو استفاده می‌کنند، بر عوارض جانبی *f ()* متکی نیستند. در نتیجه، با این راهنما مطابقت دارند.

### ۱,۳,۳ نمونه‌کد ناسازگار

در این مثال، عبارت *a ++* ارزیابی نشده‌است. در نتیجه، ارزش *a*، که پس از *b* مقداردهی اولیه شده، برابر با ۱۴ است.

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

### ۲,۳,۳ راه‌حل سازگار (*sizeof*)

در این راه‌حل، افزایش مقدار متغیر *a* خارج از عملگر *sizeof()* انجام می‌پذیرد.

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

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

### ۳,۳,۳ نمونه‌کد ناسازگار (*decltype*)

در این مثال، عبارت $i$++ داخل *decltype()* ارزیابی نشده‌است. درنتیجه، مقدار $i$ همان صفر باقی می‌ماند.

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

### ۴,۳,۳ راه‌حل سازگار (*decltype*)

در این راه‌حل، افزایش مقدار متغیر $i$، خارج از عملگر *decltype()* انجام می‌پذیرد تا ارزیابی آن ممکن گردد.

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

### ۵,۳,۳ استثنائات

**EXP52-CPP-EX1**: یک عبارت با اثرات جانبی، این اجازه را دارد تا به‌عنوان یک عملوند غیرقابل ارزیابی در تعاریف سطح بالاتر یا مفهوم SFINAE استفاده شود. اگرچه، این وضعیت‌ها برای تولید کد معتبر، به تاثیرات جانبی بستگی داشته و به مقدار تولیدشده، به‌عنوان نتیجه‌ی اثرات جانبی نیز، بستگی ندارد.

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

```
void small(int x);
void large(long long x);
#define m(x)                                    \
do                                              \
{
        if (sizeof(x) == sizeof(int))           \
        {
                small(x);                       \
        }
        else if (sizeof(x) == sizeof(long long))\
        {
                large(x);                       \
        }                                       \
} while (0)
void f()
{
        int i = 0;
        m(++i);
}
```

گسترش *m* در سطح بالاتر منجر خواهد شد تا *i*++ به‌عنوان یک عملوند ارزیابی‌نشده برای *sizeof()* استفاده شود. اگرچه، انتظار برنامه‌نویس در جایگاه توسعه و گسترش این است که *i* تنها یک‌بار در این حالت افزایش یابد.

کد زیر، نمونه‌کد سازگاری است که از یک عملوند ارزیابی‌نشده در مضمون SFINAE استفاده می‌کند تا تعیین نماید که آیا یک نوع می‌تواند به‌صورت پسوندی افزایش یابد یا خیر. در مثال *is_incrementable* ، استفاده از عملگر افزایش پسوند، تاثیرات جانبی بیشتری را تولید می‌کند تا تعیین نماید که آیا این نوع، به‌صورت پسوندی قابل افزایش هستند یا خیر. اگرچه، مقداری که از این تاثیرات جانبی به‌دست می‌آید، دور انداخته می‌شود. بنابراین، تاثیرات جانبی، تنها برای SFINAE استفاده می‌شوند.

```
#include <iostream>
#include <type_traits>
#include <utility>
template <typename T>
class is_incrementable
{
        typedef char one[1];
        typedef char two[2];
```

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه‌ای

```
        static one &is_incrementable_helper (decltype (std::declval <typename std::remove_cv
        <T>::type&>()++) *p);
        static two &is_incrementable_helper(...);
        public:
                static const bool value = sizeof(is_incrementable_helper(nullptr)) == sizeof(one);
};
void f()
{
        std::cout << std::boolalpha << is_incrementable<int>::value;
}
```

### ۶.۳.۳ ارزیابی خطر

اگر عبارت‌هایی، که برای تولید تاثیرات جانبی ظاهر می‌شوند، یک عملوند ارزیابی‌نشده باشند، ممکن است نتایج، متفاوت از آن چیزی که انتظار می‌رود، باشد. نتایج، بسته به این‌که چگونه استفاده می‌شوند، می‌توانند منجر به رفتار غیرعمدی در برنامه گردند.

| قانون | شدت/دقت | احتمال | هزینه‌ی اصلاح | اولویت | سطح |
|---|---|---|---|---|---|
| EXP52-CPP | پایین | بعید | پایین | P3 | L3 |

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

## ۴٫۳ EXP53-CPP حافظه‌ی مقداردهی اولیه‌نشده را نخوانید

اگر متغیرهای محلی خودکار، پیش از مقداردهی اولیه خوانده شوند، مقدار غیرقابل انتظاری را به خود می‌گیرند. مقداردهی اولیه‌ی پیش‌فرض شی $T$ عبارت است از:

- اگر $T$ یک کلاس است، سازنده‌ی پیش‌فرض $T$ فراخوانی می‌شود. در صورتی‌که $T$ فاقد سازنده‌ی پیش‌فرض یا نتایج سربار، مبهم بوده، یا در تابعی قرار داشته‌باشد که حذف شده‌است، و یا مقداردهی اولیه‌ی آن در دسترس نباشد، آنگاه مقداردهی اولیه، بدفرم می‌شود.
- اگر $T$ یک آرایه است، هر مؤلفه‌ی آن به‌صورت پیش‌فرض مقداردهی اولیه می‌شوند.
- در غیر این‌صورت، مقداردهی اولیه صورت نمی‌پذیرد.

در نتیجه، لازم است شی $T$ حاوی دوره‌ی ذخیره‌سازی خودکار یا پویا، منحصرا، پیش از آن‌که مقدار آن به‌عنوان بخشی از کلاس خوانده شود، مقداردهی اولیه شود. مگر آن‌که $T$ یک کلاس، یا یک آرایه، و یا به‌عبارتی، کاراکتر باریک فاقد علامت باشد؛ که در هر دو مورد، مقدار نامشخصی را نتیجه می‌دهد. این روش می‌تواند بدون این‌که منجر به رفتار نامشخصی شود، جهت اجرای عملیات کپی، از قبیل *stf::memcpy ()*، استفاده گردد.

همچنین، حافظه‌ای که به‌صورت پویا و توسط اصطلاح *new* تخصیص داده می‌شود، زمانی‌که *new-initialized* حذف می‌گردد، به‌صورت پیش‌فرض مقداردهی اولیه می‌شود. حافظه‌ی اختصاص‌یافته توسط تابع کتابخانه‌ای *std::calloc ()* به صفر مقداردهی اولیه می‌شود. حافظه‌ی تخصیصی توسط تابع کتابخانه‌ای *std::realloc ()* تصور می‌کند که مقادیر، مربوط به اشاره‌گر اصلی هستند. اما، ممکن است محدوده‌ی کامل حافظه را مقداردهی اولیه نکند. در مورد سایر روش‌های تخصیص حافظه (*std::malloc ()*، اشیای تخصیص، *operator new ()*، و سایر موارد مشابه) فرض بر آن است که مقداردهی اولیه به‌صورت پیش‌فرض انجام می‌پذیرد.

اشیای ایستا یا دوره‌ی ذخیره‌سازی نخ، پیش از آن‌که هیچ مقداردهی اولیه‌ی دیگری صورت پذیرد، به صفر مقداردهی اولیه می‌شوند و نیازی نیست پیش از آن‌که مقدار آنها خوانده شود، صراحتا مقداردهی اولیه شوند.

خواندن متغیرهای مقداردهی اولیه‌نشده جهت ایجاد آنتروپی، مسأله‌ساز است، زیرا این دسترسی‌ها به حافظه می‌توانند از طریق بهبود مفسر حذف شوند. VU#925211 نمونه‌ای از یک آسیب‌پذیری ناشی از این خطای کد است.

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

### ۱.۴.۳ نمونه‌کد ناسازگار

در این مثال، یک متغیر محلی فاقد مقدار اولیه، به‌عنوان بخشی از یک عبارت که باید مقدار آن چاپ شود، ارزیابی می‌گردد، که رفتار نامشخصی را نتیجه خواهد داد.

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

### ۲.۴.۳ راه‌حل سازگار

در این راه‌حل، شی، پیش از چاپ مقدار خود، مقداردهی اولیه می‌شود.

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

### ۳.۴.۳ نمونه‌کد ناسازگار

در این مثال، یک شی *int* * به‌وسیله‌ی یک عبارت *new* تخصیص داده شده، اما حافظه‌ای که به آن اشاره می‌کند، مقداردهی اولیه نشده‌است. مقدار اشاره‌گر و نیز مقداری که به آن اشاره می‌کند، به‌عنوان یک جریان خروجی استاندارد چاپ گردیده‌است. چاپ مقدار اشاره‌گر به‌درستی تعریف شده‌است، اما تلاش برای چاپ مقدار مورد اشاره، منجر به مقداری نامعین می‌شود و رفتار نامشخصی را با خود به‌همراه خواهد داشت.

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

### ۴.۴.۳ راه‌حل سازگار

در این راه‌حل، حافظه مستقیما پیش از چاپ مقدار خود، به ۱۲ مقداردهی اولیه شده‌است.

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

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

مقداردهی اولیه‌ی یک شی تولیدشده از یک عبارت *new*، پس از تخصیص نوع، به‌وسیله‌ی پرانتز (که می‌تواند خالی نیز باشد) یا آکولاد انجام می‌پذیرد. این راه‌حل، منجر به مقداردهی اولیه‌ی مستقیم اشاره‌شده به شی می‌شود. همان‌گونه که در کدهای زیر نمایش داده شده‌است، درصورتی‌که مقداردهی اولیه، مقداری را حذف نکند، شی به صفر مقداردهی اولیه خواهد شد:

```
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
```

### ۵,۴,۳ نمونه‌کد ناسازگار

در این مثال، متغیر عضو کلاس *c* به‌صراحت توسط *actor-initializer* در تابع تخریب پیش‌فرض مقداردهی اولیه نشده‌است. با وجود این‌که متغیر محلی *s* به‌صورت پیش‌فرض مقداردهی اولیه شده‌است، استفاده از *c* در فراخوان *S::f()* منجر به ارزیابی یک شی ارزشمند نامعین می‌گردد، که آن نیز رفتار نامشخصی را با خود به‌همراه خواهد داشت.

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

### ۶,۴,۳ راه‌حل سازگار

در این راه‌حل، *S* یک سازنده‌ی پیش‌فرض است که متغیر *c* عضو کلاس را مقداردهی اولیه می‌کند.

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

```
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);
}
```

### ۷.۴.۳ ارزیابی خطر

خواندن یک متغیر فاقد مقدار اولیه، رفتار نامشخصی است که می‌تواند منجر به رفتار غیرقابل انتظاری از برنامه شود. در برخی موارد، این نقص‌های امنیتی می‌تواند امکان اجرای دلخواه کد را فراهم سازد.

| قانون | شدت/دقت | احتمال | هزینه‌ی اصلاح | اولویت | سطح |
|---|---|---|---|---|---|
| EXP53-CPP | بالا | محتمل | متوسط | P12 | L1 |

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

## ۵٫۳ EXP54-CPP پس از طول عمر یک شی، به آن دسترسی پیدا نکنید

هر شی دارای طول عمر خاصی است که در آن بازه‌ی زمانی می‌تواند به‌صورت صحیح استفاده شود. طول عمر یک شی از زمانی آغاز می‌شود که ذخیره‌سازی مناسب، کافی، و هم‌راستا با آن برای شی فراهم و نیز، مقداردهی اولیه‌ی آن کامل انجام شده‌باشد. طول عمر یک شی زمانی به پایان می‌رسد که یک تابع مخرب، در صورت وجود، برای آن شی فراخوانی شود و حافظه‌ی ذخیره‌ساز، مجددا برای آن شی استفاده و یا آزاد گردد. استفاده از یک شی یا یک اشاره‌گر به شی، مکررا خارج از بازه‌ی طول عمر آن، منجر به رفتار نامشخصی می‌شود.

### ۱٫۵٫۳ نمونه‌کد ناسازگار

در این مثال، پیش از آن‌که طول عمر اشاره‌گر آغاز گردد، از آن برای فراخوانی یک شی عضو تابع غیر ایستا استفاده می‌شود. این امر، رفتار نامشخصی را با خود به‌همراه خواهد داشت.

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

### ۲٫۵٫۳ راه‌حل سازگار

در این راه‌حل، حافظه‌ی ذخیره‌سازی، پیش از فراخوانی *S::mem_fn()* برای اشاره‌گر فراهم می‌شود.

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

یک راه‌حل سازگار بهبودیافته، نباید حافظه را به‌صورت مستقیم و پویا تخصیص دهد، بلکه در عوض می‌تواند از یک متغیر محلی خودکار برای فراهم کردن حافظه‌ی ذخیره‌سازی و انجام فرآیند مقداردهی

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

اولیه استفاده کند. اگر به یک اشاره‌گر نیاز باشد، استفاده از یک اشاره‌گر هوشمند مانند *std: :unique_ptr* می‌تواند راه‌حلی بهبودیافته تلقی شود.

### ۳،۵،۳ نمونه‌کد ناسازگار

در این مثال، پس از آن‌که طول عمر شی به پایان می‌رسد، یک اشاره‌گر به یک شی به‌وضوح به یک کلاس پایه‌ی مجازی تبدیل می‌شود. در این حالت، برنامه، رفتار نامشخصی را از خود بروز می‌دهد.

```
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);
}
```

با وجود این‌که *f()* هرگز از شی استفاده نمی‌کند، تخصیص یک پارامتر به *f()* برای بروز رفتار نامشخص، کافی است.

### ۴،۵،۳ راه‌حل سازگار

در این راه‌حل، طول عمر *s*، جهت پشتیبانی از فراخوانی تابع *f()* گسترش داده شده‌است.

```
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
        f(s);
        delete s;
}
```

### ۵،۵،۳ نمونه‌کد ناسازگار

در این مثال، آدرس متغیر محلی از تابع *f()* بازگشت داده‌می‌شود. هنگامی‌که اشاره‌گر حاصل، از تابع *h()* گذر می‌کند، تبدیل مقدار سمت چپ به راست اِعمال‌شده به متغیر *i*، منجر به رفتار نامشخصی می‌گردد.

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

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

همان‌گونه که در این مثال مشاهده می‌شود، زمانی‌که یک اشاره‌گر به یک شی با مدت زمان و دوره‌ی ذخیره‌سازی خودکار از یک تابع، بازگشت داده‌می‌شود، برخی از مفسرها یک پیغام تشخیص تولید می‌کنند.

### ۶،۵،۳ راه‌حل سازگار

در این راه‌حل، متغیر محلی بازگشتی از تابع *g ()*، به جای آن‌که دارای دوره‌ی زمانی ذخیره‌سازی پویا باشد، نوع ایستای آن را دربر می‌گیرد و طول عمر آن برای استفاده در تابع *f ()*، به اندازه‌ی کافی گسترش می‌یابد.

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

### ۷،۵،۳ نمونه‌کد ناسازگار

یک شی *std::initializer_list<>* از لیست مقداردهنده‌ی اولیه‌ای ساخته می‌شود که اجرا، آن را به یک آرایه‌ی موقت اختصاص و آن را از سازنده‌ی *std::initializer_list<>* عبور می‌دهد. طول عمر آرایه‌ی موقت مذکور مشابه سایر اشیای موقت است، مگر آن‌که مقداردهی آغازین یک شی *std::initializer_list<>* از آرایه، طول عمر آن را افزایش دهد.

در این مثال، متغیر عضو *std: :initializer_list<int>*، مبتنی بر لیست و در ساختار *ctor-initializer* مقداردهی اولیه شده‌است. تحت این شرایط، زمانی که از ساختار خارج می‌شویم، طول عمر مفهومی آرایه‌ی

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه‌ای

موقت به پایان می‌رسد. بنابراین، دسترسی به هر عنصر از *std: :initializer_list<int>*، منجر به بروز رفتار نامشخصی می‌شود.

```
#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();
}
```

### ۸.۵.۳ راه‌حل سازگار

در این راه‌حل، *std::vector<int>* جایگزین متغیر عضو *std::initializer_list<int>* می‌شود. این متغیر جایگزین، به جای آن‌که به یک منبع متصل به آرایه‌ی موقت وابسته باشد، عناصر لیست مقداردهی اولیه را به نگهدارنده‌ی اشیا کپی می‌کند.

```
#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();
}
```

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

### ۹.۵.۳ نمونه‌کد ناسازگار

در این مثال، یک شی لاندا[۲۳] در یک شی تابع ذخیره می‌شود، که بعدا اجرای لاندا نامیده می‌شود، تا مرجع ثابتی به یک مقدار به‌دست آورد. شی لاندا مقدار صحیحی را برمی‌گرداند، که این مقدار در یک شی *int* موقت ذخیره می‌شود. شی صحیح مذکور، به *const int &* محدود می‌شود و نوعی را، که توسط شی تابع مشخص شده‌است، باز می‌گرداند. اگرچه، طول عمر اشیای موقت، پس از بازگشت از فراخوانی شی تابع، افزایش نمی‌یابد. در چنین شرایطی، زمانی‌که دسترسی به مقدار حاصل میسر می‌شود، رفتار تعریف‌نشده‌ای رخ می‌دهد.

```
#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);
}
```

### ۱۰.۵.۳ راه‌حل سازگار

در این راه‌حل، شی *std::function* یک مقدار صحیح را به جای یک مقدار *const int &* بر می‌گرداند و تضمین می‌کند که این مقدار، به جای محدود شدن به مرجع موقت، کپی شده‌است. در یک راه‌حل دیگر نیز می‌توان به جای آن‌که فراخوانی شی لاندا از طریق *std::function<>* صورت پذیرد، آن را به‌صورت مستقیم فراخوانی نمود.

```
#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);
}
```

[۲۳] Lambda

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

### ۱۱.۵.۳ نمونه کد ناسازگار

در این مثال، تابع سازنده برای متغیر خودکار *s* فراخوانی نمی‌شود، زیرا بنابر دستور *goto*، اجرا از طریق اعلان متغیر محلی انجام نمی‌پذیرد. به‌دلیل آن‌که تابع سازنده فراخوانی نشده، طول عمر *s* نیز آغاز نگردیده‌است. بنابراین، فراخوانی *S::f()*، از شی، خارج از طول عمر آن استفاده نمی‌کند، که این امر منجر به رفتار تعریف‌نشده‌ای می‌شود.

```
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();
}
```

### ۱۲.۵.۳ راه‌حل سازگار

این راه‌حل تضمین می‌کند که متغیر *s*، پیش از انجام پرش محلی، به‌درستی مقداردهی اولیه شود.

```
class S
{
        int v;
        public:
                S() : v(12) {} // Non-trivial constructor
                void f();
};
void f()
{
        S s;
        // ...
        goto bad_idea;
        // ...
        bad_idea:
                s.f();
}
```

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

### ۱۳.۵.۳ نمونه‌کد ناسازگار

در این مثال، *f ()* به‌وسیله‌ی محدوده‌ای از اشیای تکراری نوع *s* فراخوانی می‌شود. این اشیا، با استفاده از *std::copy()*، به یک بافر موقت کپی می‌شوند و زمانی‌که پردازش آنها تکمیل می‌گردد، بافر موقت آزاد می‌شود. اگرچه، بافر بازگشت داده‌شده توسط *std::get_temporary_buffer()*، شامل اشیای مقداردهی اولیه داده‌شده از نوع *s* نیست. بنابراین، هنگامی‌که *std::copy()* به تکرارکننده‌ی مقصد ارجاع می‌دهد، منجر به رفتار تعریف‌نشده‌ای می‌شود. زیرا، هنوز اشیای ارجاع داده‌شده توسط تکرارکننده‌ی مقصد، طول عمر خود را شروع نکرده‌اند. در واقع، هنگامی‌که فضا به شی اختصاص داده شده، هیچ تابع سازنده یا مقداردهنده‌ی اولیه‌ای فراخوانی نشده‌است.

```
#include <algorithm>
#include <cstddef>
#include <memory>
#include <type_traits>
class S
{
	int i;
	public:
		S() : i(0) {}
		S(int i) : i(i) {}
		S(const S&) = default;
		S& operator=(const S&) = default;
};
template <typename Iter>
void f(Iter i, Iter e)
{
	static_assert(std::is_same<typename std::iterator_traits<Iter>::value_type, S>::value, "Expecting
	iterators over type S");
	ptrdiff_t count = std::distance(i, e);
	if (!count)
	{
		return;
	}
	// Get some temporary memory.
	auto p = std::get_temporary_buffer<S>(count);
	if (p.second < count)
	{
		// Handle error; memory wasn't allocated, or insufficient memory was allocated.
		return;
	}
	S *vals = p.first;
	// Copy the values into the memory.
	std::copy(i, e, vals);
	// ...
	// Return the temporary memory.
	std::return_temporary_buffer(vals);
}
```

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

### ۱,۱۳,۵,۳ جزییات پیاده‌سازی

یک پیاده‌سازی قابل استناد و منطقی از *std::get_temporary_buffer()* و *std::copy()* می‌تواند منجر به کدهایی شود که مانند مثال زیر رفتار می‌کنند (با حذف قابلیت بررسی خطاها).

```
unsigned char *buffer = new (std::nothrow) unsigned char[sizeof(S) * object_count];
S *result = reinterpret_cast<S *>(buffer);
while (i != e)
{
        *result = *i; // Undefined behavior
        ++result;
        ++i;
}
```

عمل ارجاع مجدد به *result*، یک رفتار تعریف‌نشده‌است. زیرا حافظه‌ی اشاره‌شده، یک شی از نوع *S* و در بازه‌ی طول عمر آن نیست.

### ۱۴,۵,۳ راه‌حل سازگار (*std::uninitialized_copy()*)

در این مثال، از *std::uninitialized_copy()* به جای *std::copy()* برای کپی استفاده می‌شود، و تضمین می‌کند که اشیا، به جای استفاده از ارجاع مجدد حافظه‌ی مقدار اولیه داده‌نشده، توسط *new* مقداردهی اولیه می‌شوند. جهت خلاصه‌سازی، کد تکراری از نمونه‌کد ناسازگار حذف شده‌است.

```
//...
// Copy the values into the memory.
std::uninitialized_copy(i, e, vals);
// ...
```

### ۱۵,۵,۳ راه‌حل سازگار (*std::raw_storage_iterator*)

این راه‌حل، از *std::copy()* با *std::raw_storage_iterator* به‌عنوان تکرارکننده‌ی مقصد استفاده می‌کند، که همان نتایج به‌خوبی تعریف‌شده‌ی استفاده از *std::uninitialized_copy()* را به‌دنبال خواهد داشت. همانند راه‌حل سازگار پیشین، کد تکراری جهت خلاصه‌سازی از نمونه‌کد ناسازگار حذف شده‌است.

```
//...
// Copy the values into the memory.
std::copy(i, e, std::raw_storage_iterator<S*, S>(vals));
// ...
```

### ۱۶,۵,۳ ارزیابی خطر

ارجاع به یک شی، خارج از بازه‌ی طول عمر آن، موجب می شود تا یک مهاجم بتواند کد موردنظر و دلخواه خود را اجرا کند.

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

| قانون | شدت/دقت | احتمال | هزینه‌ی اصلاح | اولویت | سطح |
|---|---|---|---|---|---|
| **EXP54-CPP** | بالا | محتمل | بالا | P6 | L2 |

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

## ۶,۳ EXP55-CPP به یک شی cv-qualified توسط یک شی cv-unqualified دسترسی پیدا نکنید

یک نوع *const* را به‌منظور تلاش برای برای تغییر و اصلاح شی حاصل، دور نیندازید. نوع *const* بیان‌گر آن است که علی‌رغم آن‌که شاید شی، قابل اصلاح باشد، طراح رابط برنامه‌ی کاربردی (API) قصد ندارد آن را اصلاح نماید. یک نوع دارای شرایط فرار (*volatile*) را به‌منظور تلاش برای تغییر واصلاح شی حاصل دور نیندازید. نوع ثابت نشان می‌دهد که طراح رابط برنامه کاربردی (API) در نظر دارد که شی به روش‌های ناشناخته، در دسترس مفسر قرار گیرد. هرگونه دسترسی به اشیای حاصل، منجر به بروز رفتار نامشخص می‌شود.

### ۱,۶,۳ نمونه‌کد ناسازگار

در این مثال، تابع *g ()* به *const int &*، که در قالب یک متغیر صحیح قرار گرفته و اصلاح شده‌است، اشاره دارد. به‌دلیل آن‌که مقدار مورد ارجاع از قبل به‌عنوان یک ثابت اعلان شده‌است، عمل تخصیص منجر به بروز رفتار نامشخص می‌گردد.

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

### ۲,۶,۳ راه‌حل سازگار

در این راه‌حل، تابع *g ()* به *int &* اشاره دارد و برای اشاره به یک متغیر *int*، که می‌تواند اصلاح شود، به یک فراخوانی نیاز است.

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

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه‌ای

### ۳,۶,۳ نمونه‌کد ناسازگار

در این مثال، یک روش با شرایط ثابت، که برای ذخیره‌سازی پنهان نتایج ناشی از دورانداختن شرایط ثابت مذکور تلاش می‌کند، فراخوانی می‌شود. به‌دلیل آن‌که کلاس *S* قبلا به‌عنوان یک ثابت اعلام شده‌است، تغییر ناگهانی و جهش مقدار پنهان ذخیره‌شده، منجر به بروز رفتار نامشخص می‌گردد.

```
#include <iostream>
class S
{
	int cachedValue;
	int compute_value() const; // expensive
	public:
		S() : cachedValue(0) {}
		// ...
		int get_value() const
		{
			if (!cachedValue)
			{
				const_cast<S *>(this)->cachedValue = compute_value();
			}
			return cachedValue;
		}
};
void f()
{
	const S s;
	std::cout << s.get_value() << std::endl;
}
```

### ۴,۶,۳ راه‌حل سازگار

این راه‌حل، هنگام اعلان مقدار پنهان ذخیره‌شده، از کلمات کلیدی تغییرپذیر استفاده می‌کند، که به این مقدار اجازه می‌دهد که در مضمون *const*، بدون این‌که منجر به بروز رفتار نامشخص گردد، قابل تغییر باشد.

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

```
#include <iostream>
class S
{
	mutable int cachedValue;
	int compute_value() const; // expensive
	public:
		S() : cachedValue(0) {}
		// ...
		int get_value() const
		{
			if (!cachedValue)
			{
				cachedValue = compute_value();
			}
			return cachedValue;
		}
};
void f()
{
	const S s;
	std::cout << s.get_value() << std::endl;
}
```

### ۵,۶,۳ نمونه‌کد ناسازگار

در این مثال، خاصیت فرار بودن s کنار گذاشته می‌شود. تلاشی که جهت خواندن در *g ()* صورت پذیرفته، منجر به رفتار تعریف‌نشده‌ای است.

```
#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));
}
```

### ۶,۶,۳ راه‌حل سازگار

در این راه‌حل، فرض بر آن است که s باید فرار باشد. بنابراین، *g ()* اصلاح می‌شود تا *volatile S &* را بپذیرد.

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

```
#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);
}
```

### ۷,۶,۳ استثنائات

**EXP55-CPP-EX1**: یکی از استثنائات این قوانین زمانی است که باید *const* را، به هنگام استناد به یک رابط برنامه کاربردی، که پارامتر ثابتی را قبول نمی‌کند، و نیز تابعی که تلاشی برای تغییر و اصلاح متغیر مورد ارجاع انجام نمی‌دهد، دور انداخت. با این‌وجود، همیشه ترجیح داده می‌شود که رابط برنامه کاربردی تا زمانی که ممکن است به تابع صحیح *const* تغییر داده شود. برای مثال، در کد زیر، ویژگی *const* متعلق به INVFNAME در فراخوانی تابع *audit_log()* نادیده گرفته می‌شود.

```
// Legacy function defined elsewhere - cannot be modified; does not attempt to modify the contents of the passed
//parameter.
void audit_log(char *errstr);
void f()
{
	const char INVFNAME[] = "Invalid file name.";
	audit_log(const_cast<char *>(INVFNAME));
}
```

### ۸,۶,۳ ارزیابی خطر

اگر شی به‌عنوان ثابت اعلان شود، ممکن است به هنگام اجرا، در حافظه‌ی نوشتاری قرار داشته‌باشد. تلاش برای تغییر این شی، منجر به پایان غیرطبیعی برنامه یا حمله‌ی انکار خدمات می‌شود. اگر شی به‌عنوان فرار اعلان شود، مفسر قادر نخواهد بود هیچگونه فرضی برای دسترسی به شی داشته باشد. نادیده گرفتن خاصیت فرار یک شی می‌تواند منجر به خواندن یا نوشتن آن شی شود، که مرتب‌سازی مجدد یا حذف کامل آن را در پی خواهد داشت و در نهایت منجر به بروز اجرای غیرطبیعی برنامه می‌گردد.

| قانون | شدت/دقت | احتمال | هزینه‌ی اصلاح | اولویت | سطح |
|---|---|---|---|---|---|
| EXP55-CPP | متوسط | محتمل | متوسط | P8 | L2 |

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

## ۷.۳ EXP56-CPP یک تابع را توسط یک پیوند زبانی نامنطبق فراخوانی نکنید

زبان C++، درجه‌ای از تعامل را با سایر زبان‌ها، از طریق استفاده از ویژگی‌های پیوندی زبان مقدور می‌سازد. این ویژگی‌ها بر ایجاد راه‌های فراخوانی توابع و یا دسترسی به داده‌ها تاثیر می‌گذارند. به یکسری پیاده‌سازی‌ها برای ایجاد پیوندی زبانی میان C و C++ نیاز است، اما سایر پیوندهای زبانی، از قبیل جاوا، Ada، و فورترن، با مفاهیم مبتنی بر پیاده‌سازی وجود دارند.

اگر هنگام فراخوانی یک تابع، نوع تابع با تعریف آن از نظر ویژگی پیوند زبانی تطابق نداشته باشد، منجر به بروز رفتار نامشخص می‌شود. برای مثال، عدم تطابق در ویژگی پیوند زبانی می‌تواند، با توجه به قراردادهای فراخوانی یا دیگر قوانین عدم تطابق، منجر به تغییر یا خرابی پشته‌ی فراخوانی شود.

یک تابع را توسط نوعی، که پیوند زبانی آن با پیوند زبانی فراخوانی تعریف تابع مطابقت ندارد، فراخوانی نکنید. این محدودیت هم به فراخوانی توابع داخل برنامه‌ی C++ و هم به اشاره‌گرهای تابع، که جهت فراخوانی تابع از خارج از برنامه‌ی C++ استفاده می‌شوند، اعمال می‌گردد.

اگرچه، باوجود الزام ادغام پیوند زبانی در نوع تابع در برنامه‌های C++ استاندارد، بسیاری از مفسرها نمی‌توانند این کار را انجام دهند. برای مثال، تعریف مجدد تابع *f()* بجای یک سربار خوش‌فرم از تابع *f()*، قطعه کد زیر را در GCC 6.1.، Clang 3.9 و مایکروسافت ویژوال استودیو ۲۰۱۵ به یک قطعه کد بدفرم تبدیل کرده‌است.

```
typedef void (*cpp_func)(void);
extern "C" typedef void (*c_func)(void);
void f(cpp_func fp) {}
void f(c_func fp) {}
```

### ۱.۷.۳ نمونه‌کد ناسازگار

در این مثال، تابع *call_java_fn_ptr()* انتظار دارد که یک اشاره‌گر، تابع را از پیوند زبانی جاوا دریافت کند. زیرا، اشاره‌گر تابع، توسط مفسر جاوا برای بازگشت فراخوانی به کد C++ استفاده می‌شود. اگرچه، هنگامی‌که مفسر جاوا مبادرت به فراخوانی اشاره‌گر می‌نماید، تابعی که به آن با پیوند زبانی C++ اشاره شده‌است، منجر به رفتار نامشخص می‌شود. این کد، نوعی کد بدفرم است، زیرا نوع تابع فراخوان با نوع تابع فراخوانی که در جاوا استفاده می‌شود، متفاوت است. اگرچه، ممکن است با توجه به واگرایی پیاده‌سازی مشترک در استاندارد C++، برخی مفسرها به اشتباه با این کد موافقت کنند.

مدیریت مرکز امداد و هماهنگی عملیات رایانه‌ای

```
extern "java" typedef void (*java_callback)(int);
extern void call_java_fn_ptr(java_callback callback);
void callback_func(int);
void f()
{
        call_java_fn_ptr(callback_func);
}
```

### ۲.۷.۳ راه‌حل سازگار

دراین راه‌حل، جهت تطابق با *java_callback*، پیوند زبانی جاوا به تابع *callback_func()* داده شده‌است.

```
extern "java" typedef void (*java_callback)(int);
extern void call_java_fn_ptr(java_callback callback);
extern "java" void callback_func(int);
void f()
{
        call_java_fn_ptr(callback_func);
}
```

### ۳.۷.۳ ارزیابی خطر

به‌طور کلی، ویژگی عدم تطابق پیوند زبانی در پیوند زبانی میان C و C++، آسیب‌پذیری امنیتی قابل سوءاستفاده‌ای ایجاد نمی‌کند. اگرچه، پیوند زبانی در زبان‌های دیگر، که رفتار نامشخص را در پی دارد، بیشتر منجر به اجرای غیرطبیعی برنامه و آسیب‌پذیری قابل سوءاستفاده می‌شود.

| قانون | شدت/دقت | احتمال | هزینه‌ی اصلاح | اولویت | سطح |
|---|---|---|---|---|---|
| EXP56-CPP | پایین | بعید | متوسط | P2 | L3 |

## ۸,۳ EXP57-CPP اشاره‌گرهای کلاس‌های ناقص را تبدیل و یا حذف نکنید

تلاش نکنید که اشاره‌گر یک شی نوع ناقص را حذف کنید. اگرچه، در صورتی‌که کلاس فاقد مخرب کوچک اما بااهمیت باشد، و یا تابع آزادسازی مرتبطی نداشته‌باشد، آن کلاس، خوش‌فرم خواهد بود. البته، اگر بعدا مخرب حیاتی و تابع آزادسازی مذکور اضافه شوند، رفتار نامشخصی مشاهده خواهد شد. می‌توان با استفاده از انواع *static_assert* و *std::is_trivially_destructible*، مخرب ارزشمند را در زمان تفسیر بررسی نمود. اما، چنین امکانی برای بررسی تابع آزادسازی وجود ندارد.

*reinterpret_cast* ترتیبی از عملیات‌های *static_cast* است. تبدیلات سبک C یک اشاره‌گر به یک شی ناقص، یا به‌عنوان *static_cast* و یا به‌عنوان *reinterpret_cast* تعریف می‌شود (نحوه‌ی انتخاب مشخص نیست). اشاره‌گر را به نوع ناقص تبدیل نکنید. عملیات تبدیل، به‌نوبه‌ی خود، خوش‌فرم است، اما در صورتی‌که downcast (تبدیل از اشاره‌گر کلاس پایه به اشاره‌گر اشتقاقی) نتواند به چندوراثتی تنظیم شود، ممکن است ارجاع مجدد اشاره‌گر حاصل، منجر به رفتار تعریف‌نشده‌ای شود.

### ۱,۸,۳ نمونه‌کد ناسازگار

دراین مثال، یک کلاس سعی دارد اصطلاح جوش[۲۴] را پیاده‌سازی کند، اما اشاره‌گری به نوع کلاس ناقص را حذف می‌کند. اگر *Body* حاوی تابع مخرب کوچک اما بااهمیت باشد، این امر می‌تواند منجر به بروز رفتار نامشخص شود.

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

### ۲,۸,۳ راه‌حل سازگار (حذف)

در این راه‌حل، حذف *impl* به بخشی از کد که *Body* تعریف شده، منتقل شده‌است.

[۲۴] Pimpl idiom

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

### ۳.۸.۳ راه‌حل سازگار

در این راه‌حل، جهت تخصیص حافظه به *impl*، از یک *std::shared_ptr* استفاده می‌شود. برخلاف *std::unique_ptr*، یک *std::shared_ptr* قادر است به یک نوع ناقص ارجاع دهد.

```
#include <memory>
class Handle
{
        std::shared_ptr<class Body> impl;
        public:
                Handle(); ~Handle() {}
                // ...
};
```

### ۴.۸.۳ نمونه‌کد ناسازگار

ممکن است تبدیل اشاره‌گر downcast به تنظیم آدرس اشاره‌گر توسط یک مقدار ثابت، نیاز داشته‌باشد، که تنها هنگامی مشخص می‌شود که طرح و لایه‌بندی ساختار وراثتی کلاس شناخته‌شده باشد. در این مثال، *f()* یک اشاره‌گر چندریختی نوع کامل *B* را از *get_d()* بازیابی می‌کند. سپس، پیش از آن‌که اشاره‌گر مذکور به *g()* گذر کند، به یک اشاره‌گر ناقص *D* تبدیل می‌شود. ممکن است تبدیل به یک اشاره‌گر کلاس اشتقاقی، در تنظیم مناسب اشاره‌گر خلل ایجاد کد و هنگامی‌که با فراخوانی *d->do_something()*، به اشاره‌گر ارجاعی نمی‌شود، منجر به بروز رفتار نامشخص گردد.

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه‌ای

```
// 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;
}
```

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

### ۱,۴,۸,۳ جزئیات پیاده‌سازی

وقتی برنامه توسط نرم‌افزار Clang 3.8 تفسیر و تابع *f()* اجرا می‌شود، نمونه‌کد ناسازگار، خروجی زیر را چاپ می‌کند:

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

به‌همین‌ترتیب، هنگام اجرا در ویژوال استودیو ۲۰۱۵ و نیز GCC 6.1.0. ، مقدار دور از انتظاری چاپ می‌شود.

### ۵,۸,۳ راه‌حل سازگار

در این مثال، فرض بر آن است که جزئیات پیاده‌سازی توسط نوع کلاس ناقص پنهان شوند. به‌جای *D* * که به *g()* گذر کند، به نوع *B* * نیاز است.

```
// 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;
}
```

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

### ۶,۸,۳ ارزیابی خطر

تبدیل اشاره‌گرها و یا مراجع به کلاس‌های ناقص می‌تواند شرایط نامناسبی را نتیجه دهد. اگر کلاس ناقصی فاقد مخرب باشد، حذف یک اشاره‌گر از آن کلاس، رفتار تعریف‌نشده‌ای، از قبیل خاتمه‌ی برنامه، سیگنال زمان‌اجرا، و نشت منابع را در پی خواهد داشت.

| قانون | شدت/دقت | احتمال | هزینه‌ی اصلاح | اولویت | سطح |
|---|---|---|---|---|---|
| EXP57-CPP | متوسط | بعید | متوسط | P4 | L3 |

مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

## ۹,۳ EXP58-CPP از یک شی نوع درست به *va_start* گذر کنید

هنگامی‌که قانون DCL50-CPP، ساخت چنین توابعی را ممنوع می‌کند، مادامی‌که این توابع دارای پیوند خارجی زبان C هستند، می‌توانند تعریف شوند. در این وضعیت، باید هنگام فراخوانی ماکروی *va_start()*، مراقب بود. ماکروی کتابخانه‌ای استاندارد C متعلق به *va_start()*، چند محدودیت معنایی را بر نوع مقدار پارامتر دوم خود تحمیل می‌کند. تفاوت اصلی میان الزامات معنایی به شرح زیر است:

- نباید یک مرجع را به‌عنوان پارامتر دوم، از *va_start()* بگذرانید.
- گذر یک شی از کلاسی که دارای تابع سازنده‌ی کپی کوچک بااهمیت، یک تابع سازنده‌ی انتقال کوچک اما مهم، و یا یک تابع مخرب کوچک بااهمیت (به‌عنوان پارامتر دوم *va_start()*) است، به‌صورت مشروط و با معانی تعریف‌شده‌ی پیاده‌سازی پشتیبانی می‌گردد.
- باید یک پارامتر اعلان‌شده را با کلمه‌ی کلیدی *register* یا با یک نوع تابع، گذر دهید.
- گذر یک شی از یک آرایه، منجر به یک رفتار نامشخص در C++ می‌شود. زیرا، یک نوع آرایه به‌عنوان یک پارامتر تابع، نیازمند استفاده از یک مرجع بوده، که ممنوع است. علاوه براین، هنوز عبور یک شی از نوعی که تحت پوشش پارامتر پیش‌فرض است، منجر به رفتار نامشخص در C++ می‌شود.

### ۱,۹,۳ نمونه‌کد ناسازگار

در این مثال، گذر شی به *va_start()* تحت پوشش پارامتر پیش‌فرض، منجر به رفتار نامشخص می‌شود.

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

### ۲,۹,۳ راه‌حل سازگار

در این راه‌حل، تابع *f()* از *double* به جای *float* استفاده می‌کند.

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

### ۳,۹,۳ نمونه‌کد ناسازگار

در این مثال، یک نوع مرجع، به‌عنوان پارامتر دوم، به *va_start()* گذر می‌کند.

```
#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);
}
```

### ۴,۹,۳ راه‌حل سازگار

در این راه‌حل، یک نوع اشاره‌گر به جای یک نوع مرجع، به *f()* گذر می‌کند.

```
#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);
}
```

مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

### ۵,۹,۳ نمونه‌کد ناسازگار

در این مثال، یک کلاس حاوی یک تابع سازنده‌ی کپی کوچک بااهمیت (*std::string*)، به‌عنوان پارامتر دوم به *va_start()* گذر داده می‌شود، که پشتیبانی آن به‌صورت مشروط، به پیاده‌سازی وابسته است.

```
#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);
}
```

### ۶,۹,۳ راه‌حل سازگار

در این راه‌حل، یک *const char* * به جای یک *std::string*، گذر داده می‌شود و رفتار صحیحی بر تمامی پیاده‌سازی‌ها دارد.

```
#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);
}
```

### ۷,۹,۳ ارزیابی خطر

گذر یک شی نوع پشتیبانی‌نشده به‌عنوان پارامتر دوم *va_start ()*، می‌تواند رفتار تعریف‌نشده‌ای را نتیجه دهد و نقض جامعیت داده را به‌همراه داشته‌باشد.

| قانون | شدت/دقت | احتمال | هزینه‌ی اصلاح | اولویت | سطح |
|---|---|---|---|---|---|
| EXP58-CPP | متوسط | بعید | متوسط | P4 | L3 |

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ا

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

## ۱۰٫۳ EXP59-CPP از *offsetof()* در ارتباط با انواع و اعضای معتبر استفاده کنید

ماکروی *offsetof()* به‌عنوان یک شیوه‌ی قابل حمل در استاندارد C، در مبنای بایت و برای تعیین انحراف[۲۵]، از آغاز شی تا یک عضو داده‌شده‌ای از آن شی، تعریف شده‌است.

هنگام تعیین نوع پارامتر برای *offsetof()*، از کلاس لایه‌ای استاندارد استفاده کنید. هنگام تعیین پارامتر طراحی عضو برای *offsetof()*، یک فیلد بیتی، عضو داده‌ای ایستا، یا عضو تابع را گذر ندهید. گذر یک نوع یا عضو معتبر به *offsetof()*، معادل با رفتاری نامشخص است.

### ۱٫۱۰٫۳ نمونه کد ناسازگار

در این مثال، یک نوع کلاس لایه‌ای غیراستاندارد، به *offsetof()* گذر داده می‌شود، که یک رفتار نامشخص را در پی دارد.

```
#include <cstddef>
struct D
{
        virtual void f() {}
        int i;
};
void f()
{
        size_t off = offsetof(D, i);
        // ...
}
```

#### ۱٫۱٫۱۰٫۳ جزئیات پیاده‌سازی

وقتی این کد در ویژوال استودیو ۲۰۱۵ یک سیستم‌عامل ویندوز ۳۲ بیتی اجرا می‌شود، به هنگام تفسیر توسط سوئیچ *Wall/*، هیچ تشخیصی را منتشر نمی‌کند. در نتیجه، به‌دلیل حضور یک جدول مجازی از نوع *D*، مقدار *off* برابر با ۴ خواهد بود.

### ۲٫۱۰٫۳ راه‌حل سازگار

در این راه‌حل، غیرممکن است که انحراف به *i* را داخل *D* تعیین کنیم، زیرا *D* یک کلاس لایه‌ای استاندارد نیست. با این‌وجود، ساخت یک کلاس لایه‌ای استاندارد داخل *D*، امکان‌پذیر است و همان‌گونه که در این راه‌حل نشان داده شده‌است، این قابلیت برای برنامه‌ی کاربردی مهم تلقی می‌شود.

[۲۵] Offset

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

```
#include <cstddef>
struct D
{
	virtual void f() {}
	struct InnerStandardLayout
	{
		int i;
	} inner;
};
void f()
{
	size_t off = offsetof(D::InnerStandardLayout, i);
	// ...
}
```

### ۳,۱۰,۳نمونه‌کد ناسازگار

در این مثال، انحراف به *i* به‌گونه‌ای محاسبه شده‌است که یک مقدار می‌تواند در آن و داخل *buffer* ذخیره شود. اگرچه، به این دلیل که *i* یک عضو داده‌ای ایستا به‌شمار می‌رود، این مثال منجر به رفتار نامشخصی می‌گردد.

```
#include <cstddef>
struct S
{
	static int i;
	// ...
};
int S::i = 0;
extern void store_in_some_buffer( void *buffer, size_t offset, int val);
extern void *buffer;
void f()
{
	size_t off = offsetof(S, i);
	store_in_some_buffer(buffer, off, 42);
}
```

*۱,۳,۱۰,۳ جزئیات پیاده‌سازی*

وقتی این کد در ویژوال استودیو ۲۰۱۵ یک سیستم‌عامل ویندوز ۳۲ بیتی اجرا می‌شود، به هنگام تفسیر توسط سوئیچ *Wall/* هیچ تشخیصی منتشر نمی‌شود. در نتیجه، مقدار *off* عدد بزرگی خواهد بود، که نمایانگر انحراف میان آدرس اشاره‌گر تهی ۰ و آدرس متغیر ایستای *S::i* است.

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

### ۴,۱۰,۳ راه‌حل سازگار

به‌دلیل آن‌که عضوهای داده‌ای ایستا بخشی از لایه‌ی کلاس نیستند، بلکه در عوض مستقل هستند، این راه‌حل، آدرس متغیر عضو ایستا را به‌عنوان بافر برای ذخیره‌ی داده‌ها در خود، و صفر را نیز برای انحراف در نظر می‌گیرد.

```
#include <cstddef>
struct S
{
        static int i;
        // ...
};
int S::i = 0;
extern void store_in_some_buffer(void *buffer, size_t offset, int val);
void f()
{
        store_in_some_buffer(&S::i, 0, 42);
}
```

### ۵,۱۰,۳ ارزیابی خطر

گذر یک نوع یا عضو نامعتبر به *offsetof ()* می‌تواند رفتار نامشخصی را به‌همراه داشته باشد و عواقب جبران‌ناپذیری، از قبیل نقض جامعیت داده‌ها یا حصول مقادیر نادرست از ماکروها را به‌بار آورد.

| قانون | شدت/دقت | احتمال | هزینه‌ی اصلاح | اولویت | سطح |
|---|---|---|---|---|---|
| EXP59-CPP | متوسط | بعید | متوسط | P4 | L3 |

مدیریت مرکز امداد و هماهنگی رخدادهای رایانه ای

## ۱۱,۳ EXP60-CPP شی نوع لایه‌ای غیراستاندارد را از مرزهای اجرا عبور ندهید

نوع لایه‌ای استاندارد می‌تواند برای ارتباط با کدهای نوشته‌شده در سایر زبان‌ها استفاده شود. استاندارد C++، کلاس لایه‌ای استانداردی را به‌عنوان کلاسی تعریف می‌کند که دارای شرایط زیر است:

- توابع مجازی ندارد.
- کنترل دسترسی یکسانی برای تمامی اعضای داده‌ای غیرایستا وجود دارد.
- فاقد کلاس پایه‌ای هم‌نوع به‌عنوان اولین عضو داده‌ای غیرایستا است.
- دارای عضوهای داده‌ای غیرایستای اعلام‌شده‌ای است که تنها در یک کلاس داخل ساختار سلسله‌مراتبی وجود دارند.
- به‌صوررت بازگشتی، فاقد عضو داده‌ای غیرایستا از نوع لایه‌ای غیراستاندارد است.

یک محدوده‌ی اجرایی، محدوده‌ای میان کد تفسیرشده در مفسرهای متفاوت بوده، که شامل نسخه‌های گوناگونی از مفسرهای تولیدشده توسط یک فروشنده است. برای مثال، ممکن است یک تابع در یک فایل سرآیند اعلان شده، اما تعریف آن در کتابخانه‌ای، که در زمان اجرا بارگذاری شده‌است، صورت گرفته‌باشد. محدوده‌ی اجرایی بین سایت فراخوانی در محیط قابل اجرا و پیاده‌سازی تابع کتابخانه وجود دارد. همچنین، برخی از محدوده‌ها، مرز ABI (رابط باینری برنامه کاربردی) نامیده می‌شوند، زیرا به همکاری با باینری‌های برنامه‌ی کاربردی مربوط می‌شوند.

هیچ فرضی در مورد لایه‌ای خاص از اشیا با انواع لایه‌ای غیراستاندارد ایجاد نکنید. برای اشیای تفسیرشده توسط یک مفسر، که به‌وسیله‌ی کد تفسیرشده از مفسر دیگر ارجاع داده می‌شوند، برخی مفروضات منجر به نگرانی در مورد صحت و قابل حمل بودن می‌گردد. یکسان بودن لایه‌ی شی تفسیرشده توسط مفسر نخست با مفسر دوم تضمین نمی‌شود، حتی اگر هر دو مفسر منطبق بر پیاده‌سازی C++ باشند. اگرچه، ممکن است برخی پیاده‌سازی‌ها تضمین سازگاری باینری را مستند سازند، که می‌تواند مبتنی گذر اشیای لایه‌ای غیراستاندارد بین مرزهای اجرایی باشد.

یک نمونه‌ی خاص از این راهنما، شامل کد غیر C++ تفسیرشده توسط یک مفسر متفاوت است، مانند پیاده‌سازی کتابخانه‌ی استاندارد C که ازطریق کتابخانه استاندارد C++ نمایش داده می‌شود. توابع کتابخانه‌ای استاندارد C با امضاهای C++ نمایش داده می‌شوند و سیستم نوع نیز، به‌طور مداوم در حصول اطمینان از مناسب بودن و هماهنگی نوع همکاری، یاری می‌رساند. این فرآیند، امکان عبور یک اشاره‌گر را به تابعی از یک شی نوع *char* در C++، بدون هیچ کار اضافی در راستای جلوگیری از ناهماهنگی نوع،

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

فراهم نمی‌کند. با این‌حال، برخی توابع کتابخانه‌ای استاندارد C برای هر نوع اشاره‌گر C++، *void* * را می‌پذیرند، که کافی خواهد بود. گذر یک اشاره‌گر به یک نوع لایه‌ای غیراستاندارد در این وضعیت، منجر به رفتار نامشخصی می‌شود. زیرا، این امر به صحت رفتار سایر زبان‌ها در لایه‌ی شی، وابسته است.

یک شی نوع لایه‌ای غیراستاندارد را تنها هنگامی در مرزهای اجرا گذر دهید که هردو طرف مرز، به یک رابط باینری برنامه‌ی کاربردی، متصل باشند. موضوعاتی از قبیل این‌که اگر یک نسخه از مفسر برای تفسیر هر دو طرف مرز استفاده شود، یا این‌که هردو مفسر استفاده‌شده در دو طرف مرز اجرا، از رابط باینری برنامه‌ی کاربردی سازگار با نسخه‌های چندگانه استفاده کنند، یا اگر مفسرهای متفاوت از رابط باینری برنامه‌ی کاربردی یکسان استفاده نمایند، قابل قبول و مجاز است.

### ۱.۱۱.۳ نمونه‌کد ناسازگار

دراین مثال فرض بر آن است که یک کتابخانه با سرآیند library.h و یک برنامه‌ی کاربردی (به نام application.cpp) وجود دارند که از لحاظ رابط باینری برنامه‌ی کاربردی سازگار نیستند. بنابراین، محتوای library.h یک مرز اجرا را تشکیل می‌دهد. شی نوع *S* از نوع لایه‌ای غیراستاندارد است که در این مرز اجرا، گذر می‌کند. برنامه‌ی کاربردی، مثالی از یک شی از این نوع را می‌سازد و سپس، ارجاعی را به یک شی از تابع تعریف‌شده توسط کتابخانه‌ای، که در مرز اجرایی قرار دارد، گذر می‌دهد. به‌دلیل آن‌که این لایه از هیچ تضمینی برای سازگاربودن در دو سمت مرز برخوردار نیست، رفتار غیرقابل انتظاری روی می‌دهد.

```
// library.h
struct S
{
	virtual void f()
	{
		/* ... */
	}
};
void func(S &s); // Implemented by the library, calls S::f()
// application.cpp
#include "library.h"
void g()
{
	S s;
	func(s);
}
```

اگر کتابخانه و برنامه‌ی کاربردی، با یک رابط باینری برنامه‌ی کابردی یکسان مطابقت داشته باشند، این مثال به یک مثال سازگار تبدیل خواهد شد.

### ۲.۱۱.۳ راه‌حل سازگار

به‌دلیل آن‌که کتابخانه و برنامه‌ی کاربردی، با یک رابط باینری برنامه‌ی کاربردی یکسان مطابقت ندارند، این راه‌حل، یک نوع لایه‌ای استاندارد را برای کار کردن با آن مشخص و تعریف می‌کند. همچنین، یک *static_assert()* به کار گرفته می‌شود تا از *S* در برابر تغییرات آتی کد (که *S* را از نوع لایه‌ای استاندارد خارج می‌سازد)، محافظت به عمل آورد.

```
// library.h
#include <type_traits>
struct S
{
	void f()
	{
		/* ... */
	} // No longer virtual
};
static_assert(std::is_standard_layout<S>::value, "S is required to be a standard layout type");
void func(S &s); // Implemented by the library, calls S::f()
// application.cpp
#include "library.h"
void g()
{
	S s;
	func(s);
}
```

### ۳.۱۱.۳ نمونه‌کد ناسازگار

در این مثال، یک اشاره‌گر به یک شی از نوع لایه‌ای غیراستاندارد، به یک تابع که دارای پیوند زبانی فورترن است، گذر می‌کند. پیوندهای زبانی غیر از C و C++ به‌صورت مشروط به‌وسیله‌ی معانی تعریف پیاده‌سازی، پشتیبانی می‌شوند. اگر پیاده‌سازی از پیوند زبانی پشتیبانی نکند، کد، از نوع بدفرم خواهد بود. با فرض این‌که پیوند زبانی پشتیبانی می‌شود، هرگونه عملیاتی که روی شی عبوری انجام گیرد، منجر به رفتار نامشخصی می‌شود که پیامدهای امنیتی به‌دنبال خواهد داشت.

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

```
struct B
{
        int i, j;
};
struct D : B
{
        float f;
};
extern "Fortran" void func(void *);
void foo(D *d)
{
        func(d);
}
```

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

### ۴,۱۱,۳ راه‌حل سازگار

در این راه‌حل، شی نوع لایه‌ای غیراستاندارد به‌طریقی، به شی نوع لایه‌ای استاندارد محلی تبدیل شده، و سپس، به تابع فورترن گذر داده می‌شود.

```
struct B
{
        int i, j;
};
struct D : B
{
        float f;
};
extern "Fortran" void func(void *);
void foo(D *d)
{
        struct
        {
                int i, j;
                float f;
        } temp;
        temp.i = d->i;
        temp.j = d->j;
        temp.f = d->f; f
        unc(&temp);
}
```

### ۵,۱۱,۳ارزیابی خطر

تأثیرات ناشی از گذر اشیا نوع لایه‌ای غیر استاندارد در محدوده‌ی اجرایی متفاوت است و به عملیاتی که روی شی در فراخوانی و نیز، عملیاتی که توسط فراخواننده صورت می‌گیرد بستگی دارد. تأثیرات می‌توانند از درست تا رفتار نامشخص، متفاوت باشند.

| قانون | شدت/دقت | احتمال | هزینه‌ی اصلاح | اولویت | سطح |
|---|---|---|---|---|---|
| **EXP60-CPP** | بالا | محتمل | متوسط | P12 | L1 |

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

## ۱۲,۳ EXP61-CPP نباید یک شی لاندا بیشتر از اشیایی رجوع‌شده به خود، عمر کند

این امکان وجود دارد که عبارت لاندا، اشیای دارای مدت زمان ذخیره‌سازی خودکاری را از مجموعه‌ای از حوزه‌های پیوسته (حوزه‌ی دسترسی)، برای استفاده در بدنه‌ی تابع، رصد و دریافت کند. ممکن است این رصد به‌صورت صریح و از طریق مشخص کردن شی در *capture-list* لاندا، یا به‌صورت ضمنی و با استفاده از *capture-default* و ارجاع به شی در بدنه‌ی تابع لاندا صورت پذیرد. هنگام رصد صریح یا ضمنی یک شی، رصد پیش‌فرض نشان‌دهنده‌ی آن است که شی یا به‌وسیله‌ی کپی (بااستفاده از "=") و یا به‌وسیله‌ی ارجاع (بااستفاده از "&") رصد شده‌است. هنگامی‌که یک شی به‌وسیله‌ی کپی رصد می‌شود، شی لاندا شامل یک عضو داده‌ای غیرایستای بدون نام خواهد شد، که مقدار اولیه برابر با مقدار شی مورد رصد است. طول عمر عضو داده‌ای غیرایستا نیز برابر با طول عمر شی لاندا خواهد بود. با این‌وجود، هنگامی‌که یک شی توسط مرجع رصد می شود، طول عمر مورد ارجاع به طول عمر شی لاندا گره نخورده‌است.

به‌دلیل آن‌که موجودیت‌های مورد رصد، اشیایی با مدت زمان ذخیره‌سازی خودکار هستند (یا *this*)، نباید توابع بازگشتی یک شی لاندا (شامل بازگشت از طریق پارامترهای لاندا) یا ذخیره‌سازی یک شی لاندا در یک متغیر عضو یا سراسری، یک موجودیت را از طریق ارجاع رصد کنند. زیرا غالبا شی لاندا طول عمر بیشتری نسبت به مورد ارجاع و رصد دارد.

هنگامی‌که یک شی لاندا از شی مورد ارجاع و رصد خود بیشتر عمر کند، مادامی‌که شی مورد ارجاع و رصد در دسترس است، اجرای عملگر فراخوانی تابع شی لاندا منجر به بروز رفتار نامشخص می‌شود. بنابراین، شی لاندا نباید طول عمری بیشتر از شی مورد ارجاع و رصد خود داشته باشد.

### ۱,۱۲,۳ نمونه‌کد ناسازگار

دراین مثال، تابع *g()* یک لاندا را، که به‌صورت ضمنی متغیر محلی خودکار *i* را از طریق ارجاع رصد می‌کند، بر می‌گرداند. هنگامی‌که لاندا از طریق فراخوانی برگشت داده‌می‌شود، شی مورد ارجاع و رصد آن، متغیری خواهد بود که طول عمر آن به پایان رسیده‌است. در نتیجه، هنگامی‌که لاندا در *f()* اجرا می‌شود، استفاده از مرجع موجود در لاندا منجر به رفتار نامشخصی می‌گردد.

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه‌ای

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

### ۲,۱۲,۳ راه‌حل سازگار

دراین راه‌حل، لاندا شی i را نه از طریق ارجاع، بلکه از طریق کپی، رصد می‌کند. در نتیجه، لاندا شامل یک عضو داده‌ای غیرایستای ضمنی بوده که طول عمر آن، برابر با طول عمر لاندا است.

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

### ۳,۱۲,۳ نمونه‌کد ناسازگار

در این مثال، یک مرجع لاندا، یک متغیر محلی را از یک لاندای بیرونی رصد می‌کند. با این‌وجود، طول عمر این لاندای داخلی بیشتر از لاندای بیرونی و متغیرهای محلی خودکار تعریف‌شده‌ی آن است. هنگامی‌که شی لاندای داخلی در *f()* اجرا می‌شود، رفتار نامشخصی روی خواهد داد.

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

```
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();
}
```

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

### ۴,۱۲,۳ راه‌حل سازگار

دراین راه‌حل، لاندا *i* را نه از طریق ارجاع، بلکه از طریق کپی رصد می‌کند.

```
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();
}
```

### ۵,۱۲,۳ ارزیابی خطر

ارجاع یک شی خارج از طول عمر آن، این امکان را برای مهاجم فراهم می‌آورد تا کد دلخواه را اجرا نماید.

| قانون | شدت/دقت | احتمال | هزینه‌ی اصلاح | اولویت | سطح |
|---|---|---|---|---|---|
| **EXP61-CPP** | بالا | محتمل | بالا | P6 | L2 |

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

## ۱۳٫۳ EXP62-CPP به بیت‌های یک ارائه‌ی شی، که بخشی از ارائه‌ی مقدار شی نیست، دسترسی پیدا نکنید

منظور از نمایش شی[۲۶] یک شی از نوع *T*، ترتیبی از *N* شی *unsigned char* است که توسط شی *T* گرفته شده‌اند، و مقدار *N* برابر با *sizeof (T)* است. ارائه‌ی مقدار یک شی[۲۷] نیز، مجموعه بیت‌هایی است که مقدار *T* را نگه می‌دارند.

انواع کاراکتری باریک (*char*، *signed char* و *unsigned char*)، مانند سایر انواع جدایی‌ناپذیر موجود بر پلتفرم‌های خاص، دارای یک نمایش شی هستند که تنها بیت‌هایی از نمایش مقدار شی را شامل می‌شوند. برای چنین انواعی، دسترسی به هرکدام از بیت‌های نمایش مقدار، یک رفتار صحیح است. این نوع از نمایش شی به برنامه‌نویس اجازه می‌دهد که بر اساس نمایش بیت‌های یک شی، به آن دسترسی پیدا کرده و آن را تعریف کند؛ مانند فراخوانی *std::memcmp()* روی نمایش شی آن.

ممکن است سایر انواع، مانند کلاس‌ها، تنها حاوی نمایش شی متشکل از بیت‌های نمایش مقدار شی نباشند. به‌عنوان مثال، شاید دارای اعضای داده‌ای فیلد بیتی، لایه‌ی[۲۸] قرار داده‌شده میان اعضای داده‌ای، یک جدول مجازی (vtable) جهت ارسال روش مجازی، یا اعضای داده‌ای که با اولویت‌های دسترسی متنوعی اعلان شده‌اند باشند. برای چنین انواعی، بیت‌های دسترسی به نمایش شی، که بخشی از ارائه‌ی مقدار شی نیستند، بسته به آن‌که چگونه مورد دست‌یابی قرار می‌گیرند، می‌توانند رفتار تعریف‌نشده‌ای را نتیجه دهند.

به بیت‌های یک نمایش شی، که بخشی از ارائه‌ی مقدار آن شی نیستند، دسترسی پیدا نکنید. حتی اگر بیت‌ها به شیوه‌ی خوبی در دسترس هستند، مثلا از طریق آرایه‌ای از اشیای *unsigned char*، مقادیر ارائه‌شده توسط آن بیت‌ها نامشخص و یا اجرامحور هستند، و اعتماد به هر مقدار مشخصی می‌تواند منجر به اجرای غیرطبیعی برنامه گردد.

### ۱٫۱۳٫۳ نمونه‌کد ناسازگار

در این مثال، نمایش کامل شی در هنگام مقایسه‌ی دو شی از نوع *S* در دسترس است. ممکن است کلاس‌ها جهت حصول اطمینان از قرارگیری مناسب در حافظه، توسط داده‌ای، لایه‌گذاری شوند. محتوای لایه و

---

[۲۶] Object representation

[۲۷] Object's value representation

[۲۸] Pad

مقدار مازاد آن، مبتنی بر اجرا است. وقتی به جای مقایسه‌ی ارائه‌ی مقادیر، نمایش شی کلاس‌ها با یکدیگر مقایسه می‌شود، مورد مذکور می‌تواند نتایج نادرستی را به‌همراه داشته باشد. زیرا ممکن است لایی‌گذاری[۲۹] (روشی برای پر کردن بلوکی از اطلاعات با کاراکترها، اقلام، کلمات، یا رکوردهای ساختگی)، مقادیر نامشخصی را برای هر نمونه شی در نظر بگیرد.

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

### ۲.۱۳.۳ راه‌حل سازگار

در این راه‌حل، متغیر *S* باعث سربار *operator == ()* می‌شود تا مقایسه‌ای از نمایش مقدار شی انجام گیرد.

```
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)
	{
		// ...
	}
}
```

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

[۲۹] Padding

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

### ۳,۱۳,۳نمونه‌کد ناسازگار

در این مثال، *std::memset()* برای پاک کردن وضعیت داخلی یک شی استفاده می‌شود. ممکن است یک پیاده‌سازی، به‌منظور نمایش یک تابع مجازی، جدولی مجازی را داخل نمونه‌شی ذخیره کند و آن vtable، متعاقبا توسط فراخوانی *std::memset()* رونویسی شود. هنگامی‌که انتخاب و ارسال روش مجازی نیاز باشد، منجر به رفتار نامشخصی می‌شود.

```
#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
}
```

### ۴,۱۳,۳راه‌حل سازگار

در این راه‌حل، اعضای داده‌ای متغیر *S* به‌صراحت، به جای فراخوانی *std::memset()* پاک می‌شوند.

```
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
}
```

### ۵,۱۳,۳استثنائات

**EXP62-CPP-EX1**: زمانی‌که دسترسی به بیت‌های نمایش شی در کدی، که به‌خوبی تعریف شده‌است، قابل مشاهده نیست، آن دسترسی، مجاز به‌شمار می‌رود. به‌طور مشخص، هنگامی‌که هیچ اعتماد و فرضی

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

بر مقدار بیت‌هایی که بخشی از نمایش مقدار نیستند وجود ندارد، خواندن آنها مجاز است. همچنین، نوشتن بیت‌هایی که بخشی از نمایش مقدار نیستند، تنها زمانی مجاز است که آن بیت‌ها، بیت‌های لایی‌گذاری باشند. این استثنا امکان نوشتن روی بیت‌هایی را، که بخشی از نمایش شی هستند، فارغ از بیت‌های لایی‌گذاری‌شده، فراهم نمی‌کند؛ مانند جای‌نوشت یک اشاره‌گر جدول مجازی.

برای مثال، فراخوانی *std::memcpy()* روی یک شی شامل فیلد بیتی، قابل قبول است. زیرا خواندن و نوشتن بیت‌های لایی‌گذاری‌شده، قابل مشاهده نیست.

```
#include <cstring>
struct S
{
	int i : 10;
	int j;
};
void f(const S &s1)
{
	S s2;
	std::memcpy(&s2, &s1, sizeof(S));
}
```

### ۶,۱۳,۳ارزیابی خطر

تأثیرات ناشی از دست‌یابی به یک ارائه‌ی شی، که بخشی از ارائه‌ی مقدار شی محسوب نمی‌شود، می‌تواند از رفتار مبتنی بر پیاده‌سازی (مانند فرض لایه‌ای از فیلدهایی با کنترل‌های دسترسی مختلف) تا آسیب‌پذیری‌های اجرای کد (مانند جای‌نوشت اشاره‌گر vtable) متفاوت باشد.

| قانون | شدت/دقت | احتمال | هزینه‌ی اصلاح | اولویت | سطح |
|---|---|---|---|---|---|
| EXP62-CPP | بالا | محتمل | بال | P6 | L2 |

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

## ۱۴٫۳ EXP63-CPP به مقدار یک شی انتقال‌یافته تکیه نکنید

بسیاری از انواع، شامل انواع تعریف‌شده توسط کاربر و انواع تولیدشده توسط کتابخانه‌ی STL، از مفهوم انتقال پشتیبانی می‌کنند. مگر در مواردی خاص، که یک شی از نوعی که عملیات انتقال (تخصیص و مقداردهی اولیه‌ی انتقال) را پشتیبانی می‌کند، در سمت چپ یک معتبر قرار گیرد. اما وضعیت نامشخص بعد از مقدار شی، منتقل شده‌است.

گذر یک شی به‌عنوان پارامتری از تابع، که به یک پارامتر مرجع راست‌مقدار[۳۰] (rvalue) متصل است (شامل طرز فراخوانی صریح تابع، از قبیل تخصیص یا ایجاد انتقال) حالت شی را به شی دیگری منتقل می‌کند. به‌مجرد بازگشت از چنین فراخوانی تابعی، شی که به یک پارامتر مرجع راست مقدار متصل بود، به‌عنوان یک حالت *نقل‌مکان*[۳۱] در نظر گرفته می‌شود. یک‌بار که شی در این حالت قرار می‌گیرد، تنها عملیات‌هایی که ممکن است با ایمنی کافی روی نمونه‌های شی انجام گیرند، آنهایی هستند که هیچ پیش‌شرطی ندارند. زیرا، این‌که آیا این حالت از شی بتواند پیش‌شرط‌های مذکور را برآورده سازد، نامشخص است. با وجود آن‌که برخی انواع دارای پیش‌شرط‌های به‌صراحت تعریف‌شده‌ای هستند، مانند انواع تعریف‌شده توسط STL، باید فرض شود تنها عملیات‌هایی که ممکن است به‌صورت امن روی یک شی نقل‌مکانی اجرا شوند، مقداردهی‌های اولیه‌ی مجدد از طریق تخصیص به شی یا خاتمه‌ی طول عمر شی توسط مخرب آن شی هستند.

به مقدار شی نقل‌مکانی تکیه نکنید، مگر این‌که حالت نوع شی به‌خوبی مشخص شود. در واقع، تکیه بر مقادیر نامشخص منجر به رفتار تعریف‌نشده‌ای خواهد شد. به‌دلیل آن‌که نیازی به ثبت رفتار نیست، این امر می‌تواند رفتار غیرطبیعی برنامه و نیز نگرانی‌های محتمل را نتیجه دهد.

### ۱٫۱۴٫۳ نمونه‌کد ناسازگار

در این مثال، انتظار می‌رود که یک مرجع راست‌مقدار *std::string*، مقادیر ۰ تا ۹ را به‌عنوان جریان خروجی استاندارد چاپ نماید. اگرچه، به‌دلیل آن‌که شی انتقال داده می‌شود و سپس، تحت مفروضات وضعیت

---

[۳۰] Right value (rvalue)

[۳۱] Moved-from

داخلی پاک‌شده‌ی خود مجددا مورد استفاده قرار می‌گیرد، ممکن است خروجی غیر قابل انتظاری تولید شود اما رفتار نامشخصی را به‌همراه نداشته باشد.

```
#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));
	}
}
```

*۱,۱,۱۴,۳ جزئیات پیاده‌سازی*

ممکن است برخی پیاده‌سازی‌های کتابخانه‌ی استاندارد، زمانی‌که در حال اجرای *std::string* هستند، بهینه‌سازی رشته‌ی کوتاه[۳۲] (SSO) را پیاده‌سازی نمایند. درچنین پیاده‌سازی‌هایی، رشته‌های با طول مشخص در بافر کاراکتر داخلی شی *std::string* ذخیره می‌شوند تا از یک عملیات تخصیص پشته‌ی[۳۳] گران جلوگیری به‌عمل آید. اگرچه ممکن است چنین اجرایی در زمان انجام یک عملیات انتقال، مقدار اصلی بافر را تغییر ندهد. با اجرای کد فوق در Clang 3.7 و با استفاده از libc++، نتیجه‌ی زیر حاصل خواهد شد.

```
0
01
012
0123
01234
012345
0123456
01234567
012345678
0123456789
```

[۳۲] Short String Optimization (SSO)

[۳۳] Heap

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

### ۲,۱۴,۳ راه‌حل سازگار

در این راه‌حل، شی *std::string*، در هر تکرار از حلقه، به مقدار مورد انتظار مقداردهی اولیه می‌شود. این مثال تضمین می‌کند که پیش از آن‌که تلاش برای دسترسی به شی در *g()* صورت پذیرد، آن شی در یک وضعیت معتبر و مشخص قرار دارد، که خروجی مورد انتظار را به‌همراه خواهد داشت.

```
#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));
	}
}
```

### ۳,۱۴,۳ نمونه‌کد ناسازگار

در این مثال، عنصری که مقدار ۴۲ را گرفته‌است، از نگهدارنده‌ی داده‌شده حذف می‌شود. سپس، محتوایات این نگهدارنده، در جریان خروجی استاندارد چاپ می‌شوند. با این‌وجود، اگر هریک از مؤلفه‌ها از نگهدارنده حذف شوند، حلقه‌ی *for* محدوده‌محور، در محدوده‌ی تکرار نامعتبری تکرار شده، که منجر به رفتار نامشخصی می‌شود.

```
#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;
	}
}
```

### ۴,۱۴,۳ راه‌حل سازگار

در این راه‌حل، عناصر محذوف توسط الگوریتم‌های استاندارد، در طول تکرار، نادیده گرفته می‌شوند.

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

```
#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;
		}
	}
}
```

### ۵,۱۴,۳راه‌حل سازگار

در این راه‌حل، متعاقبا مؤلفه‌های محذوف توسط الگوریتم استاندارد، از نگهدارنده‌ی داده‌شده پاک می‌گردند. این روش تضمین می‌کند که حلقه‌ی مبتنی بر محدوده‌ی *for*، از یک گستره‌ی تکرار معتبر استفاده می‌نماید.

```
#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;
	}
}
```

### ۶,۱۴,۳ارزیابی خطر

عموما، حالت نقل‌مکانی شی، معتبر اما نامشخص است. تکیه بر مقادیر نامشخص می‌تواند منجر به خاتمه‌ی غیرطبیعی برنامه و نیز، نقض جامعیت داده‌ها شود.

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

| قانون | شدت/دقت | احتمال | هزینه‌ی اصلاح | اولویت | سطح |
|---|---|---|---|---|---|
| EXP63-CPP | متوسط | محتمل | متوسط | P8 | L2 |

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

# فصل چهارم

# اعداد صحیح (INT)

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

## ۱.۴ INT50-CPP به مقدار شمارشی خارج از محدوده تبدیل نکنید

شمارش‌ها[۳۴] در C++ به دو صورت هستند: مقادیر شمارشی دارای قلمرو[۳۵]، که در آنها نوع اصلی ثابت است؛ و مقادیر شمارشی بدون قلمرو، که در آن‌ها نوع اصلی می‌تواند ثابت یا متغیر باشد. محدوده‌ی مقادیری که می‌توانند توسط هر یک از شکل‌های مقادیر شمارشی نمایش داده شوند، ممکن است مقادیر شمارشگری را دربر گیرند که توسط خود شمارش مشخص نشده‌باشند.

به منظور اجتناب از انجام عملیات روی مقادیر نامشخص، مقدار ریاضیاتی که تبدیل می‌شود، باید درون بازه‌ای از مقادیری باشد که شمارش بتواند آن را نشان دهد. هنگام بررسی پویای مقادیر خارج از بازه، لازم است بررسی، پیش از عبارت تبدیل انجام گیرد.

### ۱.۱.۴ نمونه‌کد ناسازگار (بررسی کران‌ها)

این مثال، سعی دارد بررسی کند که آیا یک مقدار داده‌شده، درون بازه‌ی قابل قبولی از مقادیر شمارشی قرار دارد یا خیر. با این حال، این کار را همچنان پس از تبدیل به نوع شمارشی انجام می‌دهد، که ممکن است قادر به نمایش مقدار صحیح[۳۶] داده‌شده نباشد. در یک سیستم دو مکمله، بازه‌ی مجاز از مقادیری که می‌توانند توسط نوع شمارشی (*EnumType*) نشان داده شوند، [۰...۳] است. بنابراین، اگر مقداری خارج از این بازه به *f()* ارسال شود، تبدیل به *EnumType* منجر به یک مقدار نامشخص خواهد شد، و استفاده از این مقدار درون عبارت *if*، رفتاری نامشخص را با خود به‌همراه خواهد داشت.

```
enum EnumType
{
        First,
        Second,
        Third
};
void f(int intVar)
{
        EnumType enumVar = static_cast<EnumType>(intVar);
        if (enumVar < First || enumVar > Third)
        {
                // Handle error
        }
```

[۳۴] Enumerations
[۳۵] Scoped enumerations
[۳۶] Integer

### ۲.۱.۴ راه‌حل سازگار (بررسی کران‌ها)

این راه‌حل، نمایش مقدار را توسط نوع شمارشی و پیش از انجام تبدیل بررسی می‌کند تا تضمین نماید که تبدیل، منجر به یک مقدار نامشخص نمی‌شود. راه‌حل مذکور این کار را با محدود کردن مقدار تبدیل‌شده به مقداری که یک مقدار شمارشگر مشخص برای آن وجود دارد، انجام می‌دهد.

```
enum EnumType
{
	First,
	Second,
	Third
};
void f(int intVar)
{
	if (intVar < First || intVar > Third)
	{
		// Handle error
	}
	EnumType enumVar = static_cast<EnumType>(intVar);
}
```

### ۳.۱.۴ راه‌حل سازگار (مقدار شمارشی دارای قلمرو)

این راه‌حل، از یک مقدار شمارشی دارای قلمرو استفاده می‌کند که به‌صورت پیش‌فرض دارای یک نوع *int* اصلی است و اجازه‌ی تبدیل هر نوعی از پارامتر را به یک مقدار شمارشی معتبر می‌دهد. راه‌حل مذکور، مقدار تبدیل‌شده را به مقداری، که یک مقدار شمارشگر مشخص برای آن وجود دارد، محدود نمی‌کند. اما، همان‌گونه که در راه‌حل پیشین نشان داده‌شد، می‌تواند این کار را انجام دهد.

```
enum class EnumType
{
	First,
	Second,
	Third
};
void f(int intVar)
{
	EnumType enumVar = static_cast<EnumType>(intVar);
}
```

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه‌ای

### ۴.۱.۴ راه‌حل سازگار (مقدار شمارشی بدون قلمرو ثابت)

مشابه با راه‌حل پیشین، این راه‌حل از مقدار شمارشی بدون قلمرو استفاده می‌کند، اما یک نوع *int* اصلی ثابت فراهم می‌آورد تا هر مقداری از پارامتری که قرار است تبدیل شود، به یک مقدار شمارشی معتبر تبدیل گردد.

```
enum EnumType : int
{
        First,
        Second,
        Third
};
void f(int intVar)
{
        EnumType enumVar = static_cast<EnumType>(intVar);
}
```

همانند راه‌حل سازگار پیشین، این راه‌حل از لحاظ آن‌که با مقادیر شمارشگری، که در کد توصیف شده‌اند و نیز تبدیلات ضمنی مجاز هستند، متفاوت از نمونه‌کد ناسازگار است. راه‌حل سازگار قبلی به یک تعیین‌کننده‌ی نام تو در تو[۳۷] برای شمارشگر نیاز دارد (به‌عنوان نمونه، *EnumType::First*) و مقدار شمارشگر را به‌صورت ضمنی به *int* تبدیل نمی‌کند. مطابق با نمونه‌کد ناسازگار، این راه‌حل نیز امکان استفاده از یک تعیین‌کننده‌ی نام تو در تو را فراهم نمی‌آورد و مقدار شمارشگر را به‌صورت ضمنی به *int* تبدیل خواهد کرد.

### ۵.۱.۴ ارزیابی خطر

سربار بافر توسط مقادیر غیرمشخص، امری محتمل است و منجر به اجرای کدی دلخواه توسط یک مهاجم می‌شود. با این وجود، به‌دلیل آن‌که شمارشگرها به ندرت برای نمایه‌سازی[۳۸] به آرایه‌ها یا دیگر صورت‌های حساب اشاره‌گری[۳۹] به کار می‌روند، در مقایسه با اجرای کد دلخواه، احتمال آن‌که این سناریو منجر به نقض یکپارچگی داده‌ها[۴۰] شود، بیشتر است.

| قانون | شدت/دقت | احتمال | هزینه‌ی اصلاح | اولویت | سطح |
|---|---|---|---|---|---|
| INT50-CPP | متوسط | بعید | متوسط | P4 | L3 |

[37] Nested name specifier
[38] Indexing
[39] Pointer arithmetic
[40] Data integrity violations

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

# فصل پنجم

# کاراکترها و رشته‌ها (STR)

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

## ۱٫۵ STR50-CPP تضمین کنید که ذخیره‌سازی برای رشته‌ها، دارای فضای کافی برای داده‌های کاراکتری و کاراکتر تهی است

کپی کردن داده به بافری که به اندازه‌ی کافی برای نگهداری داده‌ها بزرگ نیست، منجر به سرریز بافر می‌شود. غالبا، سرریزهای بافر هنگام دستکاری رشته‌ها رخ می‌دهند. به‌منظور جلوگیری از چنین خطاهایی، یا لازم است کپی‌ها از طریق برش[۴۱] محدود شوند و یا ترجیحا، باید از اندازه‌ی کافی مقصد جهت نگهداری داده‌ای، که قرار است کپی شود، اطمینان حاصل نمود. رشته‌های سبک C به یک کاراکتر تهی[۴۲] برای نشان دادن انتهای رشته نیاز دارند. در حالی‌که قالب *std::basic_string* در C++، به چنین کاراکتری نیاز ندارد.

### ۱٫۱٫۵ نمونه‌کد ناسازگار

چون ورودی بدون کران است، کد زیر منجر به یک سرریز بافر می‌شود.

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

### ۲٫۱٫۵ نمونه‌کد ناسازگار

به‌منظور حل این مشکل، استفاده از روش *std::ios_base::width()* وسوسه‌کننده به‌نظر می‌رسد. اما همان‌گونه که در این مثال مشاهده می‌شود، همچنان یک تله وجود دارد.

```
#include <iostream>
void f()
{
	char bufOne[12];
	char bufTwo[12];
	std::cin.width(12);
	std::cin >> bufOne;
	std::cin >> bufTwo;
}
```

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

[41] Truncation
[۴۲] Null terminator

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

در این مثال، اولین خواندن منجر به سرریز نمی‌شود، اما می‌تواند *bufOne* را با یک رشته‌ی برش داده‌شده پُر کند. علاوه بر این، خواندن دوم هنوز می‌تواند *bufTwo* را سرریز نماید. آنگاه، operator>> یک بایت تهی (*charT()*) در مکان بعدی ذخیره می‌کند. ممکن است در صورتی‌که هیچ کاراکتری استخراج نشده باشد، این مکان، اولین مکان باشد. آنگاه operator>>، width(0) را فراخوانی می‌کند.

در نتیجه، لازم است پیش از هر فراخوانی operator>>، *width()* را فراخوانی نمود و یک آرایه‌ی کران‌دار را انتقال داد. با این‌وجود، این متعلق به ورودی برش داده‌شده نیست و ممکن است منجر به از دست دادن اطلاعات یا یک آسیب‌پذیری احتمالی شود.

### ۳.۱.۵ راه‌حل سازگار

بهترین راه‌حل برای حصول اطمینان از این‌که داده برش داده نشده‌است و نیز برای محافظت در برابر سرریزهای بافر، استفاده از *std::string* به جای یک آرایه‌ی کران‌دار است.

```
#include <iostream>
#include <string>
void f()
{
        std::string input;
        std::string stringOne, stringTwo;
        std::cin >> stringOne >> stringTwo;
}
```

### ۴.۱.۵ نمونه‌کد ناسازگار

در این مثال، تابع ورودی بدون فرمت *std::basic_istream<T>::read()*، جهت خواندن آرایه‌ی کاراکترهای فاقد فرمت از ۳۲ کاراکتر متعلق به یک فایل، استفاده می‌شود. با این‌وجود، تابع *read()* تضمین نمی‌کند که رشته با کاراکتر تهی پایان یابد، بنابراین، در صورتی‌که آرایه‌ی کاراکتری حاوی کاراکتر تهی نباشد، فراخوانی بعدی سازنده‌ی *std::string* منجر به رفتاری تعریف‌نشده می‌شود.

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

```
#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);
	// ...
}
```

### ۵,۱,۵ راه‌حل سازگار

در این راه‌حل، فرض بر آن است که ورودی فایل، حداکثر ۳۲ کاراکتر است. به جای وارد کردن کاراکتر تهی، شی *std::string* را بر مبنای تعداد کاراکترهای خواندن از جریان ورودی، می‌سازد. اگر اندازه‌ی ورودی نامعلوم[۴۳] بود، بهتر است بنابر نیاز، از *std::basic_istream<T>::readsome()* یا یک تابع ورودی فرمت‌دار، استفاده شود.

```
#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());
	// ...
}
```

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

[43] Uncertain

### ۶,۱,۵ ارزیابی خطر

کپی کردن داده‌های رشته‌ای به بافری که برای نگهداری داده‌ها خیلی کوچک است، منجر به یک سرریز بافر می‌شود. مهاجمان می‌توانند از این حالت برای اجرای کد دلخواه با مجوزهای فرآیندهای آسیب‌پذیر، سوءاستفاده کنند.

| قانون | شدت/دقت | احتمال | هزینه‌ی اصلاح | اولویت | سطح |
|---|---|---|---|---|---|
| STR50-CPP | بالا | احتمال زیاد | متوسط | P18 | L1 |

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

## ۲٫۵ STR51-CPP اقدام به ساخت یک *std::string* از یک اشاره‌گر تهی نکنید

نوع *std::basic_string* از الگوی طراحی خصیصه[۴۴] برای اداره نمودن پیاده‌سازی جزئیات انواع مختلف رشته استفاده می‌کند، که منجر به دنباله‌ای از کلاس‌های شبه رشته با یک پیاده‌سازی رایج و زیربنایی می‌شود. به‌طور مشخص، کلاس *std::basic_string* با *std::char_traits* جفت شده‌است تا کلاس‌های *std::string*، *std::wstring*، *std::u16string* و *std::u32string* را بسازد. کلاس *std::char_traits* صریحا برای فراهم نمودن جزئیات پیاده‌سازی مبتنی بر سیاست[۴۵] نوع *std::basic_string* ارائه شده‌است. چنین جزئیات پیاده‌سازی موجود در تابع *std::char_traits::length()* است، که مکررا برای تعیین تعداد کاراکترها در یک رشته با کاراکتر پایان تهی استفاده می‌شود. طبق استاندارد C++، ارسال یک اشاره‌گر تهی به این تابع، منجر به یک رفتار تعیین‌نشده می‌شود، زیرا ارجاع مجدد یک اشاره‌گر تهی را با خود به‌همراه خواهد داشت.

توابع عضو *std::basic_string* زیر منجر به فراخوانی *std::char_traits::length()* می‌شوند:

- *basic_string::basic_string(const charT *, const Allocator &)*
- *basic_string &basic_string::append(const charT *)*
- *basic_string &basic_string::assign(const charT *)*
- *basic_string &basic_string::insert(size_type, const charT *)*
- *basic_string &basic_string::replace(size_type, size_type, const charT *)*
- *basic_string &basic_string::replace(const_iterator, const_iterator, const charT *)*
- *size_type basic_string::find(const charT *, size_type)*
- *size_type basic_string::rfind(const charT *, size_type)*
- *size_type basic_string::find_first_of(const charT *, size_type)*
- *size_type basic_string::find_last_of(const charT *, size_type)*
- *size_type basic_string::find_first_not_of(const charT *, size_type)*
- *size_type basic_string::find_last_not_of(const charT *, size_type)*
- *int basic_string::compare(const charT *)*
- *int basic_string::compare(size_type, size_type, const charT *)*
- *basic_string &basic_string::operator=(const charT *)*
- *basic_string &basic_string::operator+=(const charT *)*

توابع غیرعضو *std::basic_string* زیر منجر به فراخوانی *std::char_traits::length()* می‌شوند:

- *basic_string operator+(const charT *, const basic_string&)*
- *basic_string operator+(const charT *, basic_string &&)*
- *basic_string operator+(const basic_string &, const charT *)*

---

[44] Trait
[45] Policy

- *basic_string operator+(basic_string &&, const charT *)*
- *bool operator==(const charT *, const basic_string &)*
- *bool operator==(const basic_string &, const charT *)*
- *bool operator!=(const charT *, const basic_string &)*
- *bool operator!=(const basic_string &, const charT *)*
- *bool operator<(const charT *, const basic_string &)*
- *bool operator<(const basic_string &, const charT *)*
- *bool operator>(const charT *, const basic_string &)*
- *bool operator>(const basic_string &, const charT *)*
- *bool operator<=(const charT *, const basic_string &)*
- *bool operator<=(const basic_string &, const charT *)*
- *bool operator>=(const charT *, const basic_string &)*
- *bool operator>=(const basic_string &, const charT *)*

هیچ یک از توابع فوق را با یک اشاره‌گر تهی، به‌عنوان پارامتر *const charT **  فراخوانی نکنید. این قانون، تأکید می‌کند که به اشاره‌گرهای تهی مجددا ارجاع نکنید.

### ۱,۱,۲,۵ جزئیات پیاده‌سازی

برخی از عرضه‌کنندگان کتابخانه‌های استاندارد، مانند libstdc++، هنگامی‌که از یک اشاره‌گر تهی در یکی از فراخوانی‌های توابع بالا استفاده شود، از *std::logic_error* استفاده می‌کنند، اما نه در فراخوانی *std::char_traits::length()*. با این‌وجود، *std::logic_error* برای C++ استاندارد یک نیازمندی محسوب نمی‌شود و برخی از عرضه‌کنندگان (مانند libc++ و Microsoft Visual Studio STL) این رفتار را پیاده‌سازی نمی‌کنند. نباید به‌منظور قابلیت انتقال، روی این رفتار تکیه نمود.

### ۲,۲,۵ نمونه‌کد ناسازگار

در این مثال، یک شی *std::string* از نتایج فراخوانی *std::getenv()* ساخته می‌شود. با این‌حال، چون *std::getenv()* در هنگام شکست، یک اشاره‌گر تهی برمی‌گرداند، هنگامی‌که متغیر محیطی وجود ندارد (یا برخی خطاهای دیگر رخ می‌دهد)، این کد می‌تواند منجر به رفتار تعریف‌نشده شود.

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

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

### ۳.۲.۵ راه‌حل سازگار

در این راه‌حل، نتایج حاصل از فراخوانی *std::getenv()* پیش از ساختن شی *std::string*، برای مقدار تهی بررسی می‌شوند.

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

### ۴.۲.۵ ارزیابی خطر

ارجاع مجدد به یک اشاره‌گر تهی، رفتاری تعریف‌نشده است و معمولا منجر به خاتمه‌ی غیرعادی برنامه می‌شود. با این‌وجود، در برخی موارد، ارجاع مجدد یک اشاره‌گر تهی می‌تواند منجر به اجرای کد دلخواه شود. شدت مشخص‌شده برای این حالت شدیدتر بوده، و روی پلتفرم‌هایی است که سوءاستفاده از ارجاع مجدد اشاره‌گر تهی برای اجرای کد دلخواه امکان‌پذیر نیست. شدت واقعی، کم است.

| قانون | شدت/دقت | احتمال | هزینه‌ی اصلاح | اولویت | سطح |
|---|---|---|---|---|---|
| STR51-CPP | بالا | احتمال زیاد | متوسط | P18 | L1 |

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

## ۳,۵ STR52-CPP از ارجاعات، اشاره‌گرها و تکرارکننده‌های معتبر برای ارجاع به عناصر یک *basic_string* استفاده کنید

به‌دلیل آن‌که *std::basic_string*، محفظه‌ای[46] از کاراکترها است، از ارجاعات، اشاره‌گرها و تکرارکننده‌های معتبر برای ارجاع به عناصر یک محفظه استفاده کنید. به‌عنوان یک محفظه، همانند سایر محفظه‌ها در کتابخانه‌ی STL، از تکرارکننده‌ها پشتیبانی می‌نماید. با این‌وجود، کلاس قالب *std::basic_string* دارای مفاهیم لغو غیرمعمول است. ممکن است ارجاعات، اشاره‌گرها، و تکرارکننده‌هایی که به عناصر یک دنباله‌ی *basic_string* ارجاع می‌کنند، بنابر استفاده‌های زیر از شی *basic_string*، لغو شوند:

- به‌عنوان یک پارامتر به هر تابع کتابخانه‌ی استاندارد، که به *basic_string* غیرثابت، به‌عنوان پارامتر، ارجاع می‌دهد.
- فراخوانی توابع غیرثابت، به جز عملگر []، در *front*، *back*، *begin*، *rbegin*، *end*، و *rend*.
- نمونه‌هایی از توابع کتابخانه استاندارد که به *non-const std::basic_string* ارجاع می‌دهند، شامل *std::swap()*، *::operator>>(basic_istream &, string &)*، و *std::getline()* هستند.

از یک ارجاع، اشاره‌گر یا تکرارکننده‌ی غیرمعتبر استفاده نکنید، زیرا انجام این کار منجر به رفتار تعریف‌نشده می‌شود.

### ۱,۳,۵ نمونه‌کد ناسازگار

این مثال از کد ناسازگار، *input* را در یک *std::string* کپی می‌کند و کاراکتر فضای خالی[47] (" ") را جایگزین کاراکتر نقطه ویرگول (";") می‌کند. این مثال، یک ناسازگازی است، زیرا تکرارکننده‌ی *loc* پس از اولین فراخوانی *insert()*، نامعتبر می‌شود. رفتار فراخوانی‌های بعدی *insert()*، تعریف‌نشده است.

```
#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 : ' ');
	}
}
```

[46] Container
[47] Space

### ۲،۳،۵ راه‌حل سازگار (*std::string::insert()*)

در این راه‌حل، مقدار تکرارکننده‌ی *loc* به‌عنوان نتیجه‌ی هر فراخوانی *insert()*، به‌روز رسانی می‌شود تا تکرارکننده‌ی نامعتبرشده هرگز مورد دسترسی قرار نگیرد. سپس، تکرارکننده‌ی به‌روز رسانی‌شده در انتهای حلقه افزایش می‌یابد.

```
#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 : ' ');
	}
}
```

### ۳،۳،۵ راه‌حل سازگار (*std::replace()*)

این راه‌حل، از یک الگوریتم استاندارد برای انجام جایگزینی استفاده می‌کند. هر زمان که امکان‌پذیر باشد، استفاده از یک الگوریتم عمومی برای ساخت راه‌حل ابداعی شما، ارجحیت خواهد داشت.

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

### ۴،۳،۵ نمونه‌کد ناسازگار

در این مثال، *data* پس از فراخوانی *replace()*، نامعتبر می‌شود. بنابراین، استفاده از آن در *g()* منجر به یک رفتار تعریف‌نشده خواهد شد.

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

```
#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);
}
```

### ۵,۳,۵ راه‌حل سازگار

در این راه‌حل، مادامی که اصلاحات *replace()* به پایان نرسیده‌است، اشاره‌گر به بافر داخلی *exampleString* تولید نمی‌شود.

```
#include <iostream>
#include <string>
extern void g(const char *);
void f(std::string &exampleString)
{
	// ...
	exampleString.replace(0, 2, "bb");
	// ...
	g(exampleString.data());
}
```

### ۶,۳,۵ ارزیابی خطر

استفاده از یک ارجاع، اشاره‌گر یا تکرارکننده‌ی نامعتبر به یک شی رشته‌ای، می‌تواند به یک مهاجم اجازه دهد تا کد دلخواه خود را اجرا کند.

| قانون | شدت/دقت | احتمال | هزینه‌ی اصلاح | اولویت | سطح |
|---|---|---|---|---|---|
| STR52-CPP | بالا | محتمل | بالا | P6 | L2 |

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

## ۴,۵ STR53-CPP دسترسی به مؤلفه‌ی بررسی را محدوده‌بندی کنید

در اپراتورهای نمایه‌ی *std::string*، *const_reference operator[](size_type) const* و *reference operator[](size_type)*، کاراکتر ذخیره‌شده در مکان مشخص‌شده (*pos*) را برمی‌گردانند. هنگامی‌که *pos>=size()* اجرا شود، یک ارجاع به یک شی از نوع *charT* با مقدار *charT()*، برگردانده می‌شود. عملگرهای نمایه بدون کنترل[48] هستند (هیچ استثنایی برای خطاهای بازه برقرار نیست)، و تلاش برای اصلاح شی خارج از محدوده‌ی حاصل، منجر به رفتار تعیین‌نشده می‌شود.

به‌طور مشابه، توابع *std::string::back()* و *std::string::front()* بدون کنترل هستند، زیرا برای فراخوانی از طریق *operator[]()* مناسب بدون ایجاد تعریف شده‌اند.

یک مقدار خارج از بازه را به‌عنوان پارامتر *std::string::operator[]()* نفرستید. به‌طور مشابه، *std::string::back()* یا *std::string::front()* را برای یک رشته‌ی خالی فراخوانی نکنید. تضمین کنید که نمایه‌ها و تکرارکننده‌های محفظه، درون بازه‌ی معتبر قرار دارند.

### ۱,۴,۵ نمونه‌کد ناسازگار

در این مثال، ممکن است مقدار بازگردانده شده توسط فراخوانی *get_index()*، بزرگتر از تعداد عناصر ذخیره‌شده در رشته باشد و منجر به یک رفتار تعریف‌نشده شود.

```
#include <string>
extern std::size_t get_index();
void f()
{
        std::string s("01234567");
        s[get_index()] = '1';
}
```

### ۲,۴,۵ راه‌حل سازگار (*try/catch*)

این راه‌حل، از تابع *std::basic_string::at()*، که مشابه با نمایه‌ی *operator[]* رفتار می‌کند، استفاده می‌نماید. اما، در صورتی‌که *pos >= size()* باشد، یک استثنای *std::out_of_range* ایجاد می‌کند.

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

[48] Unchecked

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

### ۳.۴.۵ راه‌حل سازگار (بررسی بازه)

این راه‌حل، پیش از فراخوانی *operator[]()*، مقدار برگردانده‌شده توسط *get_index()* را بررسی می‌کند تا اطمینان یابد که درون یک بازه‌ی معتبر قرار دارد.

```
#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
	}
}
```

### ۴.۴.۵ نمونه‌کد ناسازگار

این مثال، تلاش می‌کند تا کاراکتر اولیه در رشته را با معادل حرف بزرگ آن جایگزین کند. با این حال، اگر رشته‌ی داده‌شده تهی باشد، رفتار تعریف‌نشده مشاهده خواهد شد.

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

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه‌ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

### ۵,۴,۵ راه‌حل سازگار

در این راه‌حل، تنها زمانی *std::string::front()* فراخوانی می‌شود که رشته تهی نباشد.

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

### ۶,۴,۵ ارزیابی خطر

دسترسی بررسی‌نشده به عناصر، می‌تواند منجر به خواندن‌های خارج از بازه و سوءاستفاده‌های نوشتن و نوشتن در هرکجا[۴۹] شود. همچنین، این سوءاستفاده‌ها به‌نوبه‌ی خود می‌توانند منجر به اجرای کد دلخواه با مجوزهای فرآیندهای آسیب‌پذیر شود.

| قانون | شدت/دقت | احتمال | هزینه‌ی اصلاح | اولویت | سطح |
|---|---|---|---|---|---|
| STR53-CPP | بالا | بعید | متوسط | P6 | L2 |

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

[49] Write-anywhere

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

# فصل ششم

# مدیریت حافظه (MEM)

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

## ۱,۶ MEM50-CPP به حافظه‌ی آزاد دسترسی پیدا نکنید

ارزیابی یک اشاره‌گر (شامل ارجاع مجدد اشاره‌گر، استفاده از آن به‌عنوان یک عملوند عملیات ریاضی، تبدیل نوع آن، و استفاده از آن به‌عنوان سمت راست تخصیص) به حافظه‌ای که توسط یک تابع مدیریت حافظه آزاد شده، حاکی از یک رفتار تعریف‌نشده‌است. اشاره‌گرها به حافظه‌ای که آزاد شده‌است، اشاره‌گرهای آویزان[۵۰] نام دارند. دسترسی به اشاره‌گر آویزان می‌تواند منجر به آسیب‌پذیری‌های قابل استفاده شود.

تخصیص مجدد یا بازیافت حافظه‌ی آزاد، تحت اختیار مدیر حافظه قرار دارد. وقتی حافظه آزاد می‌شود، تمامی اشاره‌گرهایی که به آن اشاره دارند، نامعتبر خواهند شد. ممکن است این مضامین به سیستم‌عامل باز گردند، فضای آزاد را از دسترس خارج کنند، یا فعال و در دسترس باقی بمانند. در نتیجه، داده‌ی موجود در مکان آزاد می‌تواند معتبر ظاهر شود اما به‌صورت غیرمنتظره‌ای تغییر یابد. بنابراین، حافظه نباید از اولین‌باری که آزاد شده‌است، خوانده یا نوشته شود.

### ۱,۱,۶ نمونه‌کد ناسازگار (*new* و *delete*)

در این مثال، *s* پس از آن‌که آزاد می‌گردد، مجددا ارجاع داده می‌شود. اگر این دسترسی منجر به *نوشتن پس از آزاد شدن* گردد، آسیب‌پذیری می‌تواند توسط مجوزهای فرآیندهای آسیب‌پذیری، جهت اجرای کد دلخواه استفاده شود. معمولا، تخصیصات و آزادسازی‌های پویای حافظه، از حذف به دور هستند و تشخیص چنین مشکلاتی را دشوار می‌سازند.

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

[۵۰] Dangling pointers

مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکزماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

تابع *g ()* به‌عنوان *noexcept (false)* علامت زده می‌شود تا قانون MEM52-CPP (خطاهای تخصیص حافظه را شناسایی و به آنها رسیدگی کنید) را برآورده سازد.

### ۲,۱,۶ راه‌حل سازگار (*new* و *delete*)

در این راه‌حل، حافظه‌ی تخصیص‌یافته‌ی پویا، تا زمانی‌که دیگر موردنیاز نیست، آزاد نمی‌شود.

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

### ۳,۱,۶ راه‌حل سازگار (دوره‌ی ذخیره‌سازی خودکار)

در صورتی‌که شرایط ایجاب می‌کند، از دوره‌ی ذخیره‌سازی خودکار به جای دوره‌ی ذخیره‌سازی پویا استفاده کنید. به‌دلیل آن‌که نیازی نیست *s* پیش از حوزه‌ی *g ()* زنده باشد، راه‌حل مذکور از دوره‌ی ذخیره‌سازی خودکار جهت محدود نمودن طول عمر *s* در حوزه‌ی *g ()* استفاده می‌کند.

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

### ۴,۱,۶ نمونه‌کد ناسازگار (*std::unique_ptr*)

در نمونه‌کد زیر، حافظه‌ی تخصیصی پویای مدیریت‌شده توسط شی *buff*، پس از آن‌که به‌صراحت توسط مخرب شی آزاد شد، قابل دست‌یابی است.

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

```
#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;
}
```

همواره این کد، علی‌رغم این‌که از *strncpy ()* بهره می‌جوید، یک رشته‌ی بایتی مختوم به تهی ایجاد می‌کند. زیرا، *char* نهایی در مجموعه‌ی بافر را به صفر مقداردهی می‌کند.

### ۵,۱,۶ راه‌حل سازگار (*std::unique_ptr*)

در این راه‌حل، طول عمر شی *buff* نسبت به گذشته گسترش می‌یابد تا حافظه‌ی تحت مدیریت شی قابل دست‌یابی باشد.

```
#include <iostream>
#include <memory>
#include <cstring>
int main(int argc, const char *argv[])
{
	std::unique_ptr<char[]> buff;
	const char *s = "";
```

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه‌ای

```
        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;
}
```

### ۶,۱,۶ راه‌حل سازگار

در این راه‌حل، متغیری با دوره‌ی ذخیره‌سازی خودکار از نوع *std::string* به جای *std::unique_ptr<char[]>* استفاده می‌شود تا هم پیچیدگی را کاهش دهد و هم امنیت راه‌حل را بهبود بخشد.

```
#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;
}
```

### ۷,۱,۶ نمونه‌کد ناسازگار (*std::string::c_str()*)

در این مثال، *std::string::c_str()* بر یک شی *std::string* موقت فراخوانی می‌شود. اشاره‌گر منتج، زمانی به حافظه‌ی آزاد اشاره می‌کند که شی *std::string* در انتهای عبارت تخصیص، تخریب می‌شود. در نتیجه، به هنگام دست‌یابی به مؤلفه‌های اشاره‌گر، شاهد رفتار تعریف‌نشده‌ای خواهیم بود.

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه‌ای

```
#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 */
}
```

### ۸,۱,۶ راه‌حل سازگار (std::string::c_str())

در این راه‌حل، یک کپی محلی از رشته‌ی برگشتی توسط *str_func ()* ایجاد می‌شود تا تضمین نماید که استفاده از *str* زمانی معتبر خواهد بود که *display_string ()* فراخوانی شود.

```
#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 */
}
```

### ۹,۱,۶ نمونه‌کد ناسازگار

در این مثال، تلاش می‌شود بایت‌های صفر حافظه از طریق فراخوانی *operator new ()* تخصیص داده شوند. اگر این درخواست با موفقیت انجام پذیرد، به *operator new ()* جهت بازگردانی یک مقدار اشاره‌گر غیرتهی نیاز خواهد بود. اگرچه، تلاش برای تخصیص مجدد حافظه توسط چنین اشاره‌گری، رفتار تعریف‌نشده‌ای را به‌همراه خواهد داشت.

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

### ۱۰,۱,۶ راه‌حل سازگار

این راه‌حل، به هدف برنامه‌نویس بستگی دارد. اگر برنامه‌نویس قصد داشته‌باشد تا یک شی تک *unsigned char* را تخصیص دهد، راه‌حل، از *new* به جای فراخوانی مستقیم *operator new ()* استفاده خواهد کرد.

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

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

### ۱۱,۱,۶ راه‌حل سازگار

اگر برنامه‌نویس قصد داشته باشد بایت‌های صفر حافظه را تخصیص دهد، آنگاه، به جای تلاش برای ارجاع مجدد اشاره‌گر حاصل، می‌توان از اعلان *ptr* به‌عنوان یک *void* * استفاده کرد، که نمی‌تواند مجددا توسط یک پیاده‌سازی استاندارد ارجاع داده شود.

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

### ۱۲,۱,۶ ارزیابی خطر

خواندن حافظه‌ی تخصیصی پویا پس از آزادسازی، می‌تواند منجر به خاتمه‌ی غیرطبیعی برنامه و حملات انکار خدمات شود. نوشتن حافظه‌ای که آزاد شده‌است نیز می‌تواند اجرای کدهای دلخواه را با مجوزهای فرآیند آسیب‌پذیری به‌همراه داشته باشد.

| قانون | شدت/دقت | احتمال | هزینه‌ی اصلاح | اولویت | سطح |
|---|---|---|---|---|---|
| MEM50-CPP | بالا | احتمال زیاد | متوسط | P18 | L1 |

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

## ۲,۶ MEM51-CPP منابع تخصیص‌یافته را درست آزاد کنید

زبان برنامه‌نویسی C، چندین روش جهت تخصیص حافظه ارائه می‌دهد، از قبیل *std::malloc()*، *std::calloc()* و *std::realloc()*، که برنامه‌ی C++ نیز می‌تواند از آنها استفاده کند. با این‌وجود، زبان C تنها یک روش جهت آزادسازی حافظه ارائه می‌دهد: *std::free ()*.

زبان برنامه‌نویسی C++، چند روش بیشتر جهت تخصیص حافظه اضافه می‌کند، مانند عملگرهای *new*، *new[]*، جایابی *new*، و اشیای تخصیص‌گر. برخلاف C، زبان C++، چندین روش در راستای آزادسازی پویای حافظه‌ی تخصیصی ارائه می‌دهد، مانند عملگرهای *delete*، *delete[] ()*، و عملگرهای آزادسازی موجود در اشیای تخصیص‌گر.

تابع آزادسازی را روی هیچ چیزی غیر از *nullptr*، یا یک اشاره‌گر بازگشته توسط تابع تخصیص متناظر توصیف‌شده در ادامه، فراخوانی نکنید.

| تخصیص‌گر | آزادساز |
|---|---|
| *globaloperator new()/new* | *globaloperator 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()* |

گذر یک مقدار اشاره‌گر به یک تابع آزادسازی نامناسب می‌تواند منجر به رفتار تعریف‌نشده‌ای شود.

### ۱,۲,۶ نمونه‌کد ناسازگار (جایابی *new*)

در این مثال، متغیر محلی *s1* به‌عنوان عبارتی، به عملگر جایابی *new* گذر می‌کند. سپس، اشاره‌گر حاصل از این فراخوانی، به *operator delete ()::* گذر می‌کند، که رفتار تعریف‌نشده‌ای را با خود به‌همراه خواهد داشت. زیرا *operator delete ()::* در تلاش است تا حافظه‌ای را که توسط *operatr new ()::* برنگشته‌است، آزاد سازد.

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

```
#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;
}
```

### ۲,۲,۶ راه‌حل سازگار (جایابی *new*)

این راه‌حل، فراخوانی *operator delete ()*:: را حذف می‌کند، و به *s1* اجازه می‌دهد در صورت اتمام طبیعی طول عمر خود، تخریب شود.

```
#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;
	// ...
}
```

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

### ۳,۲,۶ نمونه‌کد ناسازگار (*delete* مقداردهی اولیه‌نشده)

در این مثال، تلاش می‌شود دو تخصیص داخل یک بلاک *try* اجرا شود، و اگر یکی از آنها با شکست مواجه شد، اداره‌کننده‌ی *catch*[۵۱] تلاش خواهد نمود تا منابع تخصیص‌یافته را آزاد سازد. اگرچه، به‌دلیل آن‌که متغیرهای اشاره‌گر به مقدار شناخته‌شده‌ای مقداردهی اولیه نشده‌اند، ممکن است شکست تخصیص حافظه برای *i1*، منجر به گذر *operator delete ()::* از مقداری (در *i2*) شود، که قبلا توسط فراخوانی *operator new ()::* بازنگشته‌است. در نتیجه، شاهد رفتار تعریف‌نشده‌ای خواهیم بود.

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

### ۴,۲,۶ راه‌حل سازگار (*delete* مقداردهی اولیه‌نشده)

این راه‌حل، هر دو مقدار اشاره‌گر را به *nullptr*، که مقدار معتبری برای گذر از *operator delete ()::* است، مقداردهی اولیه می‌کند.

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

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

[۵۱] Handeler

### ۵,۲,۶ نمونه‌کد ناسازگار (آرایه‌ی *new[]*)

در این مثال، یک آرایه توسط آرایه‌ی *new[]* تخصیص داده می‌شود، اما به جای فراخوانی آرایه‌ی *delete[]* جهت آزادسازی، این فرآیند را از طریق فراخوانی یک *delete* تک‌عنصری و عددی انجام می‌دهد، که رفتار تعریف‌نشده‌ای را نتیجه خواهد داد.

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

### ۶,۲,۶ راه‌حل سازگار (آرایه‌ی *new[]*)

در این راه‌حل، از فراخوانی آرایه‌ی *delete[]* به جای *delete* استفاده می‌شود.

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

### ۷,۲,۶ نمونه‌کد ناسازگار (*malloc ()*)

در این نمونه‌کد، فراخوانی *malloc ()* با فراخوانی *delete* در هم آمیخته می‌شود.

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

#### ۱,۷,۲,۶ جزئیات پیاده‌سازی

برخی پیاده‌سازی‌های *::operator new ()* منجر به فراخوانی *std::malloc ()* می‌شود. در چنین پیاده‌سازی‌هایی، لازم است تابع *::operator delete ()*، *std::free ()* را جهت آزادسازی اشاره‌گر فرا بخواند تا نمونه‌کد، به‌درستی رفتار کند.

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

### ۸,۲,۶ راه‌حل سازگار (*malloc ()*)

در این راه‌حل، اشاره‌گر تخصیص‌یافته توسط *std::malloc ()*، به جای *delete*، از طریق فراخوانی *std::free()* آزاد می‌شود.

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

### ۹,۲,۶ نمونه‌کد ناسازگار (*new*)

در این مثال، *std::free ()* جهت آزادسازی حافظه‌ای، که توسط *new* تخصیص یافته‌بود، فراخوانی می‌شود. یکی از تأثیرات جانبی رایج در رفتار تعریف‌نشده، از عدم استفاده‌ی صحیح از تابع آزادسازی نشأت می‌گیرد. در واقع، مخرب‌ها برای شیءای که توسط *std::free i()* آزاد می‌شود، فراخوانده نمی‌شوند.

```
#include <cstdlib>
struct S
{
	~S();
};
void f()
{
	S *s = new S();
	// ...
	std::free(s);
}
```

همچنین، این کد، قانون MEM53-CPP (وقتی طول عمر اشیا به‌صورت دستی مدیریت می‌شود، به‌صراحت اشیا را بسازید و از بین ببرید) را نقض می‌کند.

### ۱۰,۲,۶ راه‌حل سازگار (*new*)

در این راه‌حل، برای آزادسازی اشاره‌گر تخصیص‌یافته توسط *new*، از فراخوانی *delete* به جای فراخوانی *std::free ()* استفاده می‌شود.

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

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

### ۱۱,۲,۶ نمونه‌کد ناسازگار (کلاس *new*)

در این نمونه‌کد، عملگر عمومی *new*، توسط یک پیاده‌سازی مشخص کلاس از *operator new ()*، بازنویسی می‌شود. وقتی *new* فراخوانی می‌شود، بازنویسی کلاس مشخص انتخاب می‌شود. بنابراین، *S::operator new()* فراخوانی می‌شود. اگرچه، به‌دلیل آن‌که شی توسط یک عملگر *::delete* حوزه منهدم می‌شود، تابع *operator delete ()::* عمومی به جای پیاده‌سازی مشخص کلاس *S::operator delete ()* فراخوانی می‌شود، که رفتار تعریف‌نشده‌ای را به‌همراه خواهد داشت.

```
#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;
}
```

### ۱۲,۲,۶ راه‌حل سازگار (کلاس *new*)

در این راه‌حل، از *delete* غیرحوزه‌ای به جای *::delete* حوزه‌ای استفاده می‌شود، که منجر به فراخوانی *S::operator delete ()* خواهد شد.

```
#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;
```



مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

### ۱۳,۲,۶ نمونه‌کد ناسازگار (std::unique_ptr)

در این مثال، یک *std::unique_ptr* جهت نگهداری اشاره‌گر به یک شی، اعلان می‌شود. اما، مستقیما توسط آرایه‌ای از اشیا، مقدرادهی اولیه می‌شود. وقتی *std::unique_ptr* خراب می‌شود، حذف‌کننده‌ی پیش‌فرض آن، *delete* را به جای *delete[]* فرا می‌خواند و رفتار تعریف‌نشده‌ای را پدید می‌آورد.

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

### ۱۴,۲,۶ راه‌حل سازگار (std::unique_ptr)

در این راه‌حل، از *std::unique_ptr*، به جای اشاره‌گر به شی، جهت نگهداری آرایه‌ای از اشیا استفاده می‌شود. همچنین، *std::make_unique ()* برای مقداردهی آغازین اشاره‌گر هوشمند به کار می‌رود. در واقع، استفاده از *std::make_uniqu ()* این امکان را فراهم می‌آورد تا در صورت اشتباه بودن نوع حاصل *std::unique_ptr*، آن را تشخیص دهد.

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

### ۱۵,۲,۶ نمونه‌کد ناسازگار (std::shared_ptr)

در این نمونه‌کد، یک *std::shared_ptr* جهت نگهداری اشاره‌گر به یک شی، اعلان می‌شود. اما، مستقیما توسط آرایه‌ای از اشیا، مقدرادهی اولیه می‌شود. همانند *std::unique_ptr*، وقتی *std::shared_ptr* خراب می‌شود، حذف‌کننده‌ی پیش‌فرض آن، *delete* را به جای *[]delete* فرا می‌خواند و رفتار تعریف‌نشده‌ای را پدید می‌آورد.

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

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

### ۱۶,۲,۶ راه‌حل سازگار (*std::shared_ptr*)

برخلاف راه‌حل سازگار *std::unique_ptr*، که از *std::make_unique ()* برای ایجاد یک اشاره‌گر منحصر به‌فرد به یک آرایه استفاده می‌شد، فراخوانی *std::make_shared ()* توسط یک آرایه، بدفرم خواهد بود. در عوض، این راه‌حل، یک حذف‌کننده‌ی دستی را، که تضمین می‌کند آرایه به‌درستی حذف می‌شود، برای نوع اشاره‌گر مشترک مشخص می‌نماید.

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

### ۱۷,۲,۶ ارزیابی خطر

گذر مقدار یک اشاره‌گر به یک تابع آزادساز، که فاقد سازگاری با تابع تخصیص است، منجر به رفتار تعریف‌نشده‌ای می‌شود که می‌تواند آسیب‌پذیری‌های قابل استفاده‌ای را به‌همراه داشته باشد.

| قانون | شدت/دقت | احتمال | هزینه‌ی اصلاح | اولویت | سطح |
|---|---|---|---|---|---|
| MEM51-CPP | بالا | احتمال زیاد | متوسط | P18 | L1 |

مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

## ۳,۶ MEM52-CPP خطاهای تخصیص حافظه را تشخیص دهید و به آنها رسیدگی کنید

عملگر تخصیص حافظه‌ی پیش‌فرض، *::operator new (std::size_t)*، در صورت شکست تخصیص، یک استثنای *std::bad_alloc* ایجاد می‌کند. بنابراین، نیازی نیست بررسی نمایید که آیا فراخوانی *::operator new (std::size_t)* منجر به *nullptr* خواهد شد یا خیر. فرم غیرپرتابی، *::operator new (std::size_t, const std::nothroe_t &)*، در صورت شکست تخصیص، استثنایی ایجاد نمی‌کند، اما *nullptr* را برمی‌گرداند. همین رفتارها در ارتباط با نسخه‌های هر دو تابع تخصیص *operator new[]*، صادق است. همچنین، شی تخصیص‌گر پیش‌فرض (*std::allocator*) از *::operator new (std::size_t)* برای انجام تخصیصات استفاده می‌کند.

```
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
```

بنابراین، اگر اندازه‌ی پارامتر گذری به *new*، منفی و یا بسیار بزرگ است، *operator new[]* می‌تواند خطایی از نوع *std::bad_array_new_length*، که زیرکلاسی از *std::bad_alloc* است، ایجاد نماید.

وقتی‌که از فرم غیرپرتابی استفاده می‌کنید، الزامی است بررسی نمایید که پیش از دسترسی به اشاره‌گر حاصل، مقدار بازگشتی، *nullptr* نباشد.

### ۱,۳,۶ نمونه‌کد ناسازگار

در این مثال، آرایه‌ای از نوع *int* توسط *::operator new[](std::size_t)* ایجاد می‌شود و نتایج حاصل از تخصیص، بررسی نمی‌گردند. تابع به‌عنوان *noexcept* علامت می‌خورد، بنابراین، فراخواننده فرض می‌کند که این تابع هیچ استثنایی ایجاد نمی‌نماید. به‌دلیل آن‌که در صورت شکست تخصیص، *::operator new[](std::size_t)* می‌تواند یک استثنا ایجاد کند، خاتمه‌ی غیرطبیعی برنامه به‌وقوع خواهد پیوست.

ew[](std::size_t) می‌تواند یک استثنا ایجاد کند، خاتمه‌ی غیرطبیعی برنامه به‌وقوع خواهد پیوست.

```
#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;
}
```

### ۲,۳,۶ راه‌حل سازگار (*std::nothrow*)

وقتی از *std::nothrow* استفاده می‌شود، عملگر *new*، یا اشاره‌گر تهی و یا اشاره‌گر به فضای تخصیص را بازمی‌گرداند. همواره، اشاره‌گر را جهت حصول اطمینان از *nullptr* نبودن، پیش از ارجاع‌دهی بیازمایید. راه‌حل مذکور، شرایط خطا را به‌درستی، زمانی‌که اشاره‌گر بازگشتی *nullptr* است، بررسی می‌کند.

```
#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;
}
```

### ۳,۳,۶ راه‌حل سازگار (*std::bad_alloc*)

متناوبا، می‌توان از *::operator new[]* فاقد *std::nothrow* و در عوض، در صورتی‌که حافظه‌ی کافی برای تخصیص وجود ندارد، یک استثنای *std::bad_alloc* گرفت.

```
#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;
}
```

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

### ۴,۳,۶ راه‌حل سازگار (*noexcept(false)*)

اگر طراحی تابع به‌گونه‌ای است که انتظار می‌رود فراخواننده به شرایط استثنایی رسیدگی کند، مجاز است که تابع را به‌عنوان تابعی که ممکن است استثنایی ایجاد نماید، معرفی کنیم. موکدا، علامت‌زنی تابع نیاز نیست، زیرا عدم حضور *noexcept* در هر تابعی مبین آن است که تابع مذکور اجازه‌ی ایجاد استثنا را دارد.

```
#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;
}
```

### ۵,۳,۶ نمونه‌کد ناسازگار

در این مثال، دو تخصیص حافظه، داخل یک عبارت انجام می‌گیرند. به‌دلیل آن‌که تخصیصات حافظه به‌عنوان پارامترهایی، به فراخوانی تابع گذر می‌کنند، یک استثنای ایجادشده ناشی از یکی از فراخوانی‌های *new*، می‌تواند منجر به نشت حافظه شود.

```
struct A
{
	/* ... */
};
struct B
{
	/* ... */
};
void g(A *, B *);
void f()
{
	g(new A, new B);
}
```

شرایطی را درنظر بگیرید که در آن، ابتدا *A* تخصیص و ایجاد شده، و سپس، *B* تخصیص داده می‌شود و استثنایی را ایجاد می‌نماید. پوشاندن فراوانی *g ()* در یک بلاک *try/catch*، کافی نیست. زیرا، آزادسازی حافظه‌ی تخصیص‌یافته به *A*، غیرممکن است.

همچنین، این مثال، قانون EXP50-CPP (به ترتیب ارزیابی برای تأثیرات جانبی، وابسته نباشید) را نقض می‌کند. زیرا، ترتیب پارامترهای مورد ارزیابی *g ()*، نامشخص است.

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

### ۶,۳,۶ راه‌حل سازگار (*std::unique_ptr*)

در این راه‌حل، یک *std::unique_ptr* برای مدیریت منابع اشیای *A* و *B* از طریق RAII، استفاده می‌شود. طبق شرایط مطروحه در کد ناسازگار، *B*، که در حال ایجاد یک استثنا است، موجب تخریب و آزادسازی شی *A* می‌شود.

```
#include <memory>
struct A
{
	/* ... */
};
struct B
{
	/* ... */
};
void g(std::unique_ptr<A> a, std::unique_ptr<B> b);
void f()
{
	g(std::make_unique<A>(), std::make_unique<B>());
}
```

### ۷,۳,۶ راه‌حل سازگار (مراجع)

اگر شرایط مهیا باشد، بهترین و مطمئن‌ترین راه‌حل سازگار ممکن حذف کامل تخصیص حافظه و در عوض، گذر اشیا با استفاده از مراجع است.

```
#include <memory>
struct A
{
	/* ... */
};
struct B
{
	/* ... */
};
void g(A &a, B &b);
void f()
{
A a;
B b;
g(a, b);
}
```

### ۸,۳,۶ ارزیابی خطر

عدم موفقیت در تشخیص شکست‌های تخصیص می‌تواند منجر به خاتمه‌ی غیرطبیعی برنامه و نیز حملات انکار خدمات شود. اگر حافظه‌ی منابع آسیب‌پذیر برنامه از مقدار بازگشتی انحراف پیدا کنند، مهاجم قادر

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

خواهد بود از برنامه، جهت خواندن یا نوشتن دلخواه حافظه، سوءاستفاده نماید. این آسیب‌پذیری می‌تواند منجر به اجرای کد دلخواه شود.

| قانون | شدت/دقت | احتمال | هزینه‌ی اصلاح | اولویت | سطح |
|---|---|---|---|---|---|
| MEM52-CPP | بالا | احتمال زیاد | متوسط | P18 | L1 |

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

## ۴٫۶ MEM53-CPP وقتی طول عمر اشیا به‌صورت دستی مدیریت می‌شود، به‌صراحت اشیا را بسازید و از بین ببرید

ایجاد اشیای تخصیص‌یافته‌ی پویا در C++، در دو مرحله اتفاق می‌افتد. گام نخست، شامل تخصیص حافظه‌ی کافی جهت ذخیره نمودن شی است؛ و مرحله‌ی دوم، مسئول مقداردهی اولیه‌ی تکه‌ی جدید حافظه، بسته به نوع شی تولیدی، است.

به‌طور مشابه، تخریب اشیای تخصیص‌یافته در C++ نیز طی دو مرحله رخ می‌دهد. گام نخست، مسئولیت به‌پایان رساندن شی را، بسته به نوع شی، بر عهده دارد؛ در گام دوم نیز، فضای تخصیص‌یافته به شی، آزاد می‌شود.

طول عمر یک شی، ویژگی زمان‌اجرایی آن شی است. اگر شی، از نوع کلاس یا به‌هم پیوسته باشد، و آن شی یا یکی از اعضای آن توسط سازنده‌ای غیر از یک سازنده‌ی پیش‌فرض بدیهی مقداردهی اولیه شود، آنگاه شی مذکور از مقداردهی اولیه‌ی غیربدیهی برخوردار است. طول عمر شی $T$ زمانی آغاز می‌گردد که:

- ذخیره‌سازی با تنظیم مناسب و اندازه‌ی $T$ به‌دست می‌آید.
- اگر شی دارای مقداردهی اولیه‌ی غیربدیهی باشد، مقداردهی آغازین آن کامل است.

طول عمر شی $T$ زمانی به پایان می‌رسد که:

- اگر $T$ از نوع کلاس و نیز حاوی یک مخرب غیربدیهی باشد، فراخوانی مخرب آغاز گردد.
- ذخیره‌سازی تصرفات شی مورد استفاده‌ی مجدد قرار گیرد و یا آزاد شود.

برای یک شی تخصیص‌یافته‌ی پویا، معمولا این دو مرحله به‌صورت خودکار و با استفاده از عملگرهای *new* و *delete* انجام می‌پذیرند. اصطلاح *new T* برای شی $T$، منجر به فراخوانی *operator new ()* جهت تخصیص حافظه‌ی کافی به $T$ خواهد شد. اگر تخصیص حافظه موفقیت‌آمیز باشد، سازنده‌ی پیش‌فرض $T$ فراخوانی می‌شود. نتیجه‌ی این عبارت، یک اشاره‌گر $P$ به یک شی از نوع $T$ است. وقتی اشاره‌گر در عبارت *delete P* به کار گرفته می‌شود، منجر به فراخوانی مخرب $T$ خواهد شد. به هنگام تکمیل مخرب، *operator delete ()* جهت آزادسازی حافظه‌ی تخصیص‌یافته فرا خوانده می‌شود.

هنگامی‌که برنامه، شی تخصیص‌یافته‌ی پویا را با روش‌هایی غیر از عملگر *new* ایجاد می‌کند، گفته می‌شود که برنامه، طول عمر شی را به‌صورت دستی مدیریت نموده‌است. این شرایط، به هنگام استفاده از سایر روش‌های تخصیص، از قبیل استفاده از یک شی تخصیص‌گر یا *malloc ()* پدیدار می‌شود.

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه‌ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

در زمان مدیریت دستی طول عمر، لازم است سازنده جهت مقداردهی اولیه‌ی طول عمر شی، فرا خوانده شود. همچنین، مخرب باید جهت خاتمه بخشیدن به طول عمر، فراخوانی شود. استفاده از یک شی، خارج از طول عمر آن، رفتار تعریف‌نشده‌ای را به‌همراه خواهد داشت. یک شی می‌تواند یا از طریق فراخوانی *new* و یا با فراخوانی تابع *construct ()* ایجاد کند. همچنین، فراخوانی مخرب صریح و یا فراخوانی تابع *destroy ()*، راه‌های تخریب یک شی به‌شمار می‌روند.

### ۱.۴.۶ نمونه‌کد ناسازگار

در این مثال، کلاس با مقداردهی اولیه‌ی غیربدیهی (به‌دلیل حضور سازنده‌ی کاربرمحور)، توسط فراخوانی *std::malloc ()* ایجاد می‌شود. اگرچه، سازنده‌ی شی هرگز فراخوانی نمی‌شود. در نتیجه، بعدا که دسترسی به کلاس توسط فراخوانی *s->f* تقاضا می‌شود، شاهد رفتار تعریف‌نشده‌ای خواهیم بود.

```
#include <cstdlib>
struct S
{
	S();
	void f();
};
void g()
{
	S *s = static_cast<S *>(std::malloc(sizeof(S)));
	s->f();
	std::free(s);
}
```

### ۲.۴.۶ راه‌حل سازگار

در این راه‌حل، هر دوی سازنده و مخرب به‌صراحت فراخوانی می‌شوند. بنابراین، جهت پیشگیری از استفاده از شی، خارج از بازه‌ی طول عمر آن، ذخیره‌سازی زیربنایی، متغیری جدا از شی زنده خواهد بود.

```
#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);
}
```

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه‌ای

### ۳.۴.۶ نمونه‌کد ناسازگار

در این نمونه‌کد، یک کلاس نگهدارنده‌ی دستی از یک شی تخصیص‌گر در راستای فراهم نمودن ذخیره‌سازی انواع مؤلفه‌ای دلخواه، استفاده می‌کند. در حالی‌که فرض می‌شود تابع *copy_elements ()*، سازنده‌های کپی را برای مؤلفه‌هایی که به فضای ذخیره‌سازی تازه تخصیص‌یافته فراخوانی می‌کند، این مثال در فراخوانی صریح سازنده‌های پیش‌فرض، برای مؤلفه‌های مازادی که در حال ذخیره‌شدن هستند، با شکست مواجه می‌شود. اگر چنین مؤلفه‌ای از طریق تابع *operator[] ()* مورد دسترسی قرار گیرد، بسته به نوع *T*، شاهد رفتار تعریف‌نشده‌ای خواهیم بود.

```
#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];
	}
};
```

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

### ۴,۴,۶ راه‌حل سازگار

در این مثال، تمامی مؤلفه‌ها به‌درستی، توسط فراخوانی صریح کپی و یا سازندگان پیش‌فرض $T$، مقداردهی اولیه می‌شوند.

```
#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];
		}
};
```

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

### ۵,۴,۶ استثنائات

**MEM53-CPP-EX1**: اگر شی به بداهت قابل ساخت (تخریب) باشد، نیازی نیست تا سازنده‌ی (مخرب) آن به‌صراحت جهت مقداردهی اولیه‌ی (پایان بخشیدن به) طول عمر شی، فرا خوانده شود. این ویژگی‌ها می‌توانند توسط فراخوانی *std::is_trivially_constructible()* و *std::is_trivially_destructible()* از *<type_traits>* آزموده شوند. به‌عنوان مثال، انواع پیوسته‌ای مانند *int* و *long long*، به فراخوانی صریح سازنده یا مخرب نیازی ندارند.

### ۶,۴,۶ ارزیابی خطر

عدم موفقیت در ساخت و یا تخریب یک شی، حالت داخلی آن را ناپایدار خواهد نمود، که می‌تواند منجر به رفتار تعریف‌نشده و نیز، افشای تصادفی اطلاعات گردد.

| قانون | شدت/دقت | احتمال | هزینه‌ی اصلاح | اولویت | سطح |
|---|---|---|---|---|---|
| **MEM53-CPP** | بالا | احتمال زیاد | متوسط | P18 | L1 |

مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

## ۵٫۶ MEM54-CPP توسط اشاره‌گر مرتب مناسب، جایابی جدید را برای ظرفیت ذخیره‌سازی کافی فراهم کنید.

هنگام فراخوانی یک نوع توسط عبارت *new*، اَشکال فاقد جایابی عمومی *operator new* متعلق به C++ تلاش می‌کنند فضای ذخیره‌سازی کافی را به یک شی‌ای از یک نوع تخصیص دهند که آن شی، در صورت موفقیت، یک اشاره‌گر را با ترتیب مناسب برای هر شی با نیازمندی ترتیبی بنیادین، بازمی‌گرداند. با این‌وجود، عملگر جایابی پیش‌فرض *new*، به‌سادگی اشاره‌گر را، بدون تضمین آن‌که فضای کافی برای ساخت شی وجود دارد و یا اطمینان از این‌که اشاره‌گر، نیازمندی‌های ترتیب مناسب را ارضا می‌کند یا خیر، به فراخواننده برمی‌گرداند.

اشاره‌گری را که به‌صورت مناسب مرتب نشده است، برای شی ایجادشده متعلق به جایابی *new*، عبور ندهید. انجام این کار منجر به ساخت شی در مکانی نامرتبط و در نتیجه، منجر به رفتاری نامشخص می‌شود. اشاره‌گری را که فاقد ظرفیت ذخیره‌سازی کافی است، برای اشیای ساخته‌شده عبور ندهید. انجام این کار منجر به مقداردهی اولیه‌ی حافظه، خارج از مرز شی می‌شود که رفتار نامشخصی را با خود به‌همراه خواهد داشت.

### ۱٫۵٫۶ نمونه‌کد ناسازگار

در این مثال، اشاره‌گری به *short*، از جایابی *new* عبور داده می‌شود، که این جایابی در تلاش برای مقداردهی اولیه به *long* است. این امر، در معماری‌هایی که شرایط *sizeof(short) < sizeof(long)* در آنها صادق است، منجر به بروز رفتار نامشخص می‌شود. در این مثال و مثال‌های بعد، فرض بر آن است که پس از آن‌که طول عمر ذخیره‌سازی اشاره‌گر برگشت داده‌شده توسط جایابی *new* به پایان می‌رسد، دیگر آن اشاره‌گر، قابل استفاده نخواهد بود. برای مثال، اشاره‌گر، در یک متغیر عمومی *static* ذخیره نخواهدشد و پس از اتمام فراخوانی *f()* نیز ، مجددا ارجاع داده نخواهدشد.

```
#include <new>
void f()
{
        short s;
        long *lp = ::new (&s) long;
}
```

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه‌ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

### ۲,۵,۶ نمونه‌کد ناسازگار

این مثال تضمین می‌کند که *long* به‌عنوان بافری با اندازه‌ی کافی ساخته شده‌است. اگرچه، تضمین نمی‌کند که تمامی نیازمندی‌های ترتیبی اشاره‌گر عبور داده شده به جایابی *new* برطرف شوند. جهت شفاف‌تر ساختن نمونه‌کد زیر، یک متغیر محلی *c* نیز، اضافه و اعلان شده‌است.

```
#include <new>
void f()
{
        char c; // Used elsewhere in the function
        unsigned char buffer[sizeof(long)];
        long *lp = ::new (buffer) long;
        // ...
}
```

### ۳,۵,۶ راه‌حل سازگار (*alignas*)

در این راه‌حل، برای اطمینان از این‌که بافر به‌صورت مناسب برای *long* تنظیم شده‌است یا خیر، از یک اعلان صریح *alignas* استفاده می‌شود.

```
#include <new>
void f()
{
        char c; // Used elsewhere in the function
        alignas(long) unsigned char buffer[sizeof(long)];
        long *lp = ::new (buffer) long;
        // ...
}
```

### ۴,۵,۶ راه‌حل سازگار (*std::aligned_storage*)

این راه‌حل تضمین می‌نماید که *long* به‌عنوان بافری با اندازه‌ی کافی و نیز تنظیمی مناسب ساخته شده‌است.

```
#include <new>
void f()
{
        char c; // Used elsewhere in the function
        std::aligned_storage<sizeof(long), alignof(long)>::type buffer;
        long *lp = ::new (&buffer) long;
        // ...
}
```

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

### ۵,۵,۶ نمونه‌کد ناسازگار (شکست در محاسبه‌ی سربار آرایه)

این نمونه‌کد تلاش دارد ذخیره‌سازی کافی از ترتیب مناسب را به به آرایه‌ای از اشیای *S* اختصاص دهد. اگرچه، این مثال در محاسبه‌ی سرباری، که ممکن است یک پیاده‌سازی به مقدار ذخیره‌سازی آرایه اضافه کند، با شکست مواجه می‌شود. سربار (که معمولا به‌عنوان کوکی[۵۲] از آن یاد می‌شود) جهت ذخیره‌ی شماری از مؤلفه‌ها در آرایه (به‌گونه‌ای که عبارت حذف آرایه بتواند مخرب را برای هر مؤلفه‌ی ایجاد شده‌ی آرایه فراخوانی نماید) ذخیره کند.

```
#include <new>
struct S
{
        S (); ~S ();
};
void f()
{
        const unsigned N = 32;
        alignas(S) unsigned char buffer[sizeof(S) * N];
        S *sp = ::new (buffer) S[N];
        // ...
        // Destroy elements of the array.
        for (size_t i = 0; i != n; ++i)
        {
                sp[i].~S();
        }
}
```

### ۶,۵,۶ راه‌حل سازگار (Clang/GCC)

مقدار سربار موردنیاز عبارات جدید آرایه، نامشخص است. اما، به‌صورت ایده‌آل می‌تواند توسط کیفیت پیاده‌سازی‌ها ثبت شود.راه‌حل سازگار زیر، مختص مفسرهای Clang و GCC است و تضمین می‌نماید که سربار تخصیصات پویای آرایه، تک‌مقداری از نوع *size_t* است. همچنین، راه‌حل مذکور جهت تأمین ایمنی، عملگر جایابی *new []* را دچار سربار می‌کند تا اندازه‌ی بافر را به‌عنوان پارامتر سوم بپذیرد و نیز تأیید نماید که اندازه‌ی مذکور، حداقل به اندازه‌ی مقدار ذخیره‌سازی موردنیاز است.

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

[۵۲] Cookie

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

```
#include <cstddef>
#include <new>
#if defined(__clang__) || defined(__GNUG__)
        const size_t overhead = sizeof(size_t);
#else static_assert(false, "you need to determine the size of your implementation's array overhead");
        const size_t overhead = 0;
        // Declaration prevents additional diagnostics
        // about overhead being undefined; the value used
        // does not matter.
#endif
struct S
{
        S(); ~S();
};
void *operator new[](size_t n, void *p, size_t bufsize)
{
        if (n <= bufsize)
        {
                throw std::bad_array_new_length();
        }
        return p;
}
void f()
{
        const size_t n = 32;
        alignas(S) unsigned char buffer[sizeof(S) * n + overhead];
        S *sp = new (buffer, sizeof(buffer)) S[n];
        // ...
        // Destroy elements of the array.
         for (size_t i = 0; i != n; ++i)
        {
                sp[i].~S();
        }
}
```

### ۷.۵.۶ ارزیابی خطر

گذر نامناسب اشاره‌گرهای نامرتب یا اشاره‌گرهای که به ذخیره‌سازی ناکافی جایابی *new* اشاره دارند، می‌توان رفتارهای تعریف‌نشده‌ای، از قبیل سرریز بافر و نیز خاتمه‌ی غیرعادی برنامه، را به‌همراه داشته‌باشند.

| قانون | شدت/دقت | احتمال | هزینه‌ی اصلاح | اولویت | سطح |
|---|---|---|---|---|---|
| MEM54-CPP | بالا | احتمال زیاد | متوسط | P18 | L1 |

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

## ۶,۶ MEM55-CPP به نیازمندی‌های مدیریت ذخیره‌سازی پویای جایگزینی اولویت دهید

پیاده‌سازی‌های سفارشی می‌توانند به‌صورت عمومی، جایگزین توابع رهاسازی و تخصیص حافظه‌ی پویا شوند. به‌عنوان مثال، یک کاربر می‌تواند از حافظه‌ی پویای یک برنامه‌ی کاربری را شرح دهد و تصمیم بگیرد که تخصیص‌دهنده‌ی پیش‌فرض، به‌دلیل الگوی استفاده‌ای که دارد، بهینه نیست و یک استراتژی متفاوت تخصیص می‌تواند توسعه یابد.

بنابر شرط تابع جایگزینی، یک جایگزینی برای هر تابع رهاسازی و تخصیص حافظه‌ی پویا باید نیازمندی‌های معنایی را، که توسط رفتار موردنیاز مناسب مشخص شده‌اند، برآورده سازد.

### ۱,۶,۶ نمونه‌کد ناسازگار

در این مثال، یک پیاده‌سازی سفارشی، جایگزین تابع عمومی *operator new(std::size_t)* شده‌است. اگرچه، کارایی این پیاده‌سازی پایین‌تر از بهره‌وری تابعی است که جایگزین آن شده‌است. به‌طور مشخص، اگر تخصیص‌دهنده‌ی سفارشی نتواند مقدار حافظه‌ی درخواستی را اختصاص دهد، تابع جایگزین به جای ایجاد یک استثنا از نوع *std::bad_alloc* اشاره‌گری با مقدار *null* را برمی‌گرداند. با بازگشت یک اشاره‌گر خالی، ممکن است توابع وابسته به رفتار *operator new(std::size_t)*، به جای تخصیص حافظه، مجددا به اشاره‌گر خالی ارجاع دهند.

```
#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
```

### ۲,۶,۶ راه‌حل سازگار

هنگامی‌که تخصیص با شکست مواجه می‌شود، این راه‌حل، با ایجاد یک استثنای *std::bad_alloc* مناسب، رفتار مورد نیاز تابع تخصیص‌دهنده‌ی عمومی جایگزین‌شده را پیاده‌سازی می‌کند.

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

```
#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
```

### ۳.۶.۶ ارزیابی خطر

شکست در برآوردساختن نیازمندی‌های تابع ذخیره‌سازی پویای جایگزین منجر به رفتار تعریف‌نشده‌ای خواهد شد. در برخی موارد، ارجاع مجدد به یک اشاره‌گر خالی می‌تواند تبعات سنگینی، همچون اجرای کد دلخواه را به‌همراه داشته‌باشد.

| قانون | شدت/دقت | احتمال | هزینه‌ی اصلاح | اولویت | سطح |
|---|---|---|---|---|---|
| **MEM55-CPP** | بالا | احتمال زیاد | متوسط | P18 | L1 |

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

## ۷,۶ MEM56-CPP یک مقدار اشاره‌گر از پیش مالکیت‌یافته را در یک اشاره‌گر هوشمند غیرمرتبط ذخیره نکنید.

اشاره‌گرهای هوشمندی مانند *std::unique_ptr* و *std::shared_ptr* مالکیت معنایی اشاره‌گرها را به‌عنوان بخشی از نوع سیستم، رمزگشایی می‌کنند. آنها مقدار اشاره‌گر را می‌پوشانند، توسط توابع عضو *operator* *()* و *operator->()* معانی و مفاهیم اشاره‌گرمانند را فراهم می‌سازند، و طول عمر اشاره‌گری را که مدیریت می‌کنند کنترل می‌نمایند. وقتی اشاره‌گر هوشمندی از یک مقدار اشاره‌گر ایجاد می‌گردد، گفته می‌شود که آن مقدار به تملک اشاره‌گر هوشمند مذکور در آمده‌است.

فراخوانی *std::unique_ptr::release()* از مالکیت مقدار اشاره‌گر مدیریت‌شده، صرف‌نظر خواهد نمود. تخریب، تخصیص انتقالی، یا فراخوانی *std::unique_ptr::reset()* بر یک شی *std::unique_ptr* نیز، از مقدار اشاره‌گر مدیریت‌شده چشم‌پوشی خواهد کرد، اما منجر به تخریب این مقدار خواهد شد. همین شرایط برای شی *std::shared_ptr* نیز برقرار است.

یک اشاره‌گر هوشمند نامرتبط را توسط مقدار اشاره‌گری، که به تملک اشاره‌گر هوشمند دیگری در آمده‌است، ایجاد نکنید. فراخوانی‌هایی از این قبیل (مانند فراخوانی *reset ()*) منجر به تنظیم مجدد آن اشاره‌گر هوشمند به یک مقدار اشاره‌گر از پیش تملک‌یافته می‌شود.

### ۱,۷,۶ نمونه‌کد ناسازگار

در این مثال، دو اشاره‌گر هوشمند نامرتبط، از یک مقدار اشاره‌گر مشابه ساخته می‌شوند. هنگامی‌که متغیر خودکار محلی *p2* ازبین می‌رود، مقدار اشاره‌گری را که مدیریت می‌کند نیز حذف می‌نماید. سپس، هنگامی‌که متغیر محلی خودکار *p1* ازبین می‌رود، همان مقدار اشاره‌گر را حذف می‌کند که منجر به آسیب‌پذیری دوگانه‌ی آزاد می‌شود.

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

### ۲,۷,۶ راه‌حل سازگار

در این راه حل، اشیای *std::shared_ptr* از طریق ساختار کپی به یکدیگر وابسته‌اند. هنگامی‌که متغیر خودکار محلی *p2* ازبین می‌رود، شمار استفاده‌ی مقدار اشاره‌گر مشترک کاهش می‌یابد اما صفر نمی‌شود.

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه‌ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

سپس، هنگامی که متغیر محلی خودکار *p1* ازبین می‌رود، شمار استفاده‌ی مقدار اشاره‌گر مشترک به صفر کاهش می‌یابد و اشاره‌گر مدیریت‌شده از بین می‌رود. همچنین این راه‌حل، به جای تخصیص یک اشاره‌گر خام و ذخیره‌ی مقدار آن در متغیر محلی، *std::make_shared()* را فراخوانی می‌کند.

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

### ۳.۷.۶ ارزیابی خطر

گذر مقدار اشاره‌گر به تابع رهاسازی، که فاقد تطابق با تابع تخصیص متناظر از پیش تعیین‌شده است، رفتار تعریف‌نشده‌ای از قبیل آسیب‌پذیری‌های قابل سوءاستفاده را با خود به‌همراه خواهد داشت.

| قانون | شدت/دقت | احتمال | هزینه‌ی اصلاح | اولویت | سطح |
|---|---|---|---|---|---|
| **MEM56-CPP** | بالا | احتمال زیاد | متوسط | P18 | L1 |

مدیریت مرکز امداد هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

# فصل هفتم

# ورودی و خروجی (FIO)

مدیریت مرکز امداد هماهنگی عملیات رخدادهای رایانه ای

## ۱,۷ FIO50-CPP به‌طور متناوب و بدون مداخله‌ی یک فراخوانی مکان‌یابی، از یک جریان فایل، ورودی و خروجی نگیرید

محدودیت‌های خواندن و نوشتن یک دنباله، که توسط یک شی از کلاس *basic_filebuf<charT, traits>* کنترل می‌شوند، مشابه با خواندن و نوشتن از کتابخانه‌ی استاندارد C، FILES است. استاندارد C، محدودیت‌های زیر را روی اشیای FILE، که برای هر دو عمل خواندن و نوشتن باز شده‌اند، اعمال می‌کند:

- هنگامی که یک فایل با حالت به‌روز رسانی باز می‌شود، هم ورودی و هم خروجی می‌توانند روی جریان مربوطه اعمال شوند.
- خروجی نباید مستقیما توسط ورودی، فاقد یک فراخوانی میانی[۵۳] به تابع *flush* یا یک تابع مکان‌یابی فایل (*fseek*، *fsetpos* یا *rewind*) باشد.
- ورودی نباید مستقیما توسط خروجی و بدون یک فراخوانی میانی به یک تابع مکان‌یابی فایل دنبال شود، مگر این‌که عملیات ورودی به انتهای فایل برسد.

در نتیجه، سناریوهای زیر می‌توانند منجر به یک رفتار تعریف‌نشده شوند:

- دریافت مستقیم ورودی از یک جریان به دنبال خروجی به آن جریان، بدون یک فراخوانی میانی به *std::basic_filebuf<T>::seekoff()* (در صورتی‌که فایل به پایان نرسیده باشد).
- خروجی به یک جریان، پس از دریافت ورودی از آن جریان بدون فراخوانی *std::basic_filebuf<T>::seekoff()* (در صورتی‌که فایل به پایان نرسیده باشد).

اگر یک فراخوانی به یک تابع مکان‌یابی فایلِ کتابخانه‌ی استاندارد C انجام شود، هیچ تابع *std::basic_filebuf<T>* دیگری رفتار را تضمین نمی‌کند.

در نهایت، فراخوانی *std::basic_ostream<T>::seekp()* یا *std::basic_istream<T>::seekg()* منجر به یک فراخوانی *std::basic_filebuf<T>::seekoff()* برای مکان‌یابی جریان فایل خواهد شد. با توجه به این‌که *std::basic_iostream<T>* از *std:: basic_ostream<T>* و *std::basic_istream<T>*، و *std::fstream* از *std::basic_iostream* ارث می‌برند، هریک از توابع برای فراخوانی و حصول اطمینان از این‌که بافر فایل، قبل و بعد از عملیات I/O، در یک وضعیت معتبر قرار دارد، قابل قبول هستند.

---

[۵۳] Intervening call

### ۱.۱.۷ نمونه‌کد ناسازگار

این مثال، داده‌ها را به انتهای یک فایل اضافه می‌کند و سپس از همان فایل می‌خواند. با این وجود، به‌دلیل آن‌که هیچ فراخوانی مکان‌یابی میانی بین خورجی فرمت‌دار و فراخوانی‌های ورودی وجود ندارد، رفتار، تعریف‌نشده است.

```
#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;
}
```

### ۲.۱.۷ راه‌حل سازگار

در این راه‌حل، تابع *std::basic_istream<T>::seekg()* بین خروجی و ورودی فراخوانی شده، و منجر به حذف رفتار تعریف‌نشده می‌شود.

```
z#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); f
	ile >> str;
}
```

### ۳.۱.۷ ارزیابی خطر

ورود و خروج متناوب از یک جریان، بدون یک فراخوانی میانی *flush* یا مکان‌یابی، یک رفتار تعریف‌نشده را در پی دارد.

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

| قانون | شدت/دقت | احتمال | هزینه‌ی اصلاح | اولویت | سطح |
|---|---|---|---|---|---|
| **FIO50-CPP** | پایین | احتمال زیاد | متوسط | P6 | L2 |

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

## ۲,۷ FIO51-CPP وقتی دیگر به فایل‌ها نیاز ندارید، آنها را ببندید

یک فراخوانی به تابع *std::basic_filebuf<T>::open()* باید پیش از پایان زمان آخرین اشاره‌گری که مقدار فراخوانی را برمی‌گرداند یا قبل از خاتمه‌ی برنامه‌ی عادی (هر کدام که زودتر رخ دهد)، با یک فراخوانی به *std::basic_filebuf<T>::close()* تطبیق داده شود.

شایان ذکر است، *std::basic_ifstream<T>*، *std::basic_ofstream<T>* و *std::basic_fstream<T>*، همگی یک ارجاع داخلی به یک شی *std::basic_filebuf<T>* را، که روی آن *open()* و *close()* در هنگام نیاز فراخوانی می‌شوند، نگه می‌دارند. مدیریت مناسب یک شی از یکی از این انواع، برای حصول اطمینان از انطباق با این قانون، کافی است. اغلب، بهترین راه‌حل، استفاده از شی جریانی با مقادیر معنایی، به جای تخصیص حافظه‌ی پویا است، که انطباق با MEM51-CPP را تضمین می‌نماید. به‌طور مناسب، منابعی که به‌صورت پویا تخصیص داده شده‌اند را بازگردانید. با این‌وجود، انجام این کارها هنوز برای وضعیت‌هایی که مخرب‌ها به‌طور خودکار فراخوانی نمی‌شوند، کافی نیست.

### ۱,۲,۷ نمونه‌کد ناسازگار

در این مثال، یک شی *file* از *std::fstream* ساخته می‌شود. سازنده‌ی *std::fstream*، *std::basic_filebuf<T>::open()* را فراخوانی می‌کند. *std::terminate_handler* پیش‌فرض، که توسط *std::terminate()* فراخوانی می‌شود، *std::abort()* است که مخرب‌ها را فراخوانی نمی‌کند. در نتیجه، شی پایه‌ی *std::basic_filebuf<T>*، توسط شی‌ای که به‌صورت مناسب بسته نشده‌است، نگهداری می‌شود.

```
#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();
}
```

برنامه را ناگهان خاتمه ندهید. به‌دلیل اختصار، نمایش ماهیت مسأله به عملگر حذف شده‌است.

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

### ۲.۲.۷ راه‌حل سازگار

در این راه‌حل، *std::fstream::close()* پیش از *std::terminate()* فراخوانی شده‌است تا از بسته‌شدن مناسب منابع فایل اطمینان حاصل نماید.

```
#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();
}
```

### ۳.۲.۷ راه‌حل سازگار

در این راه‌حل، جریان پیش از آن‌که *std::terminate()* فراخوانی شود، به‌صورت ضمنی و از طریق RAII، بسته می‌شود تا از بسته‌شدن مناسب منابع فایل اطمینان حاصل کند.

```
#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();::terminate();
}
```

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

### ۴,۲,۷ ارزیابی خطر

ممکن است شکست در بستن مناسب فایل، به یک مهاجم اجازه دهد تا منابع سیستم را به کار گیرد و به اتمام برساند[۵۴]. همچنین، می‌تواند خطر ناشی از آن‌که داده‌های نوشته‌شده روی بافرهای فایل درون‌حافظه‌ای، در زمان رخداد خاتمه‌ی غیرعادی برنامه، پاک[۵۵] نشوند، را افزایش دهد.

| قانون | شدت/دقت | احتمال | هزینه‌ی اصلاح | اولویت | سطح |
|---|---|---|---|---|---|
| FIO51-CPP | متوسط | بعید | متوسط | P4 | L3 |

مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

[۵۴] Exhaust

[۵۵] Flush

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

# فصل هشتم

# اداره کردن استثنائات و خطاها (ERR)

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

## ۱.۸ ERR50-CPP به یک‌باره برنامه را خاتمه ندهید

توابع *std::abort()*، *std::quick_exit()* و *std::_Exit()* برای خاتمه بخشیدن ناگهانی برنامه استفاده می‌شوند. بنابراین این توابع، وظیفه‌ی محوله‌ی خود را بدون فراخوانی اداره‌کنندگان خروجی ثبت‌شده توسط *std::atexit()* و نیز، بدون اجرای مخرب‌های اشیایی، که دارای دوره‌های ذخیره‌سازی پویا، نخ، و یا ایستا هستند، انجام می‌دهند. این‌که چگونه یک سیستم جریانات ورودی را به هنگام خاتمه‌ی برنامه کنترل می‌کند، مبتنی بر شرایط پیاده‌سازی است. ممکن است جریانات باز[۵۶] حاوی داده‌ی بافرشده‌ی نوشته‌نشده، به‌سرعت جریان یابند و یا نیابند؛ ممکن است جریانات مذکور بسته شوند یا نشوند؛ و ممکن است فایل‌های موقتی حذف شوند یا نشوند. به‌دلیل آن‌که توابع مذکور می‌توانند منابع خارجی (مانند فایل‌ها و ارتباطات شبکه‌ای) را رها کنند، باید صراحتا در وضعیت میانی و در پاسخی مستقیم به خطایی حیاتی از برنامه، فراخوانی شوند.

تابع *std::terminate()*، تابع *terminate_handler* جاری را، که خود، به‌صورت پیش‌فرض تابع *std::abort()* را فرا می‌خواند، فراخوانی می‌کند. چندین راه جهت فراخوانی *std::terminate()* وجود دارد، که در ادامه مطرح می‌شوند:

- وقتی مکانیزم رسیدگی به استثنا، پس از اتمام مقداردهی اولیه‌ی شی استثنا و پیش از فعال‌سازی اداره‌کننده‌ی آن استثنا، تابعی را که توسط یک استثنا خروج می‌یابد، فرا می‌خواند.
- هنگامی‌که یک عبارت *throw* بدون استفاده از هیچ عملگری، تلاش می‌کند تا مجددا استثنایی را ایجاد نماید و هیچ استثنایی رسیدگی نمی‌شود.
- وقتی یک مکانیزم رسیدگی به استثنا قادر نیست اداره‌کننده‌ای را برای یک استثنای ایجادشده بیابد.
- هنگامی‌که جست‌وجو برای یک اداره‌کننده با انسداد خارجی یک تابع حاوی مشخصه‌ی *noexcept* مواجه می‌شود، که این تابع امکان ایجاد استثنا را فراهم نمی‌نماید.
- وقتی تخریب یک شی در طول پیمایش یک پشته، از طریق ایجاد یک استثنا خاتمه می‌یابد.
- هنگامی‌که مقداردهی اولیه‌ی یک متغیر غیر محلی با دوره‌ی ذخیره‌سازی ایستا یا نخ، توسط استثنایی خروج می‌یابد.

---

[۵۶] Open streams

- وقتی تخریب یک متغیر غیر محلی با دوره‌ی ذخیره‌سازی ایستا یا نخ، توسط استثنایی خروج می‌یابد.
- هنگامی‌که اجرای یک تابعی، که توسط *std::atexit()* و *std::at_quick_exit()* ثبت‌شده‌است، به‌وسیله‌ی یک استثنا خارج می‌شود.
- وقتی اداره‌کننده‌ی استثنای غیرمنتظره‌ی پیش‌فرض پیاده‌سازی، فراخوانی می‌شود.
- زمانی‌که *std::unexpected()*، استثنایی را ایجاد می‌کند، که برای مشخصه‌ی استثنای نقض‌شده‌ی پیشین مجاز نیست، و *std::bad_exception()* در مشخصه‌ی استثنای پویا فراخوانی نمی‌شود.
- هنگامی‌که *std::nested_exception::rethrow_nested()* برای شی‌ای که استثنایی را اتخاذ نکره‌است، فراخوانی می‌شود.
- وقتی اجرای تابع آغازین نخ توسط یک استثنا خروج می‌یابد.
- زمانی‌که مخرب برای شی‌ای از نوع *std::thread* فراخوانی می‌شود، که آن شی به یک نخ قابل اتصال بازمی‌گردد.
- هنگامی‌که عملگر تخصیص کپی برای شی‌ای از نوع *std::thread* فراخوانی می‌شود، که آن شی به یک نخ قابل اتصال بازمی‌گردد.
- وقتی فراخوانی *condition_variable::wait()*، *condition_variable::wait_until()*، یا *condition_variable::wait_for()* در برطرف نمودن شرایط پسین با شکست مواجه می‌شود: *lock.owns_lock() == true* یا *lock.mutex()* با فراخوانی نخ، قفل نمی‌شود.
- وقتی فراخوانی *condition_variable_any::wait()*، *condition_variable_any::wait_until()*، یا *condition_variable_any::wait_for()* در برطرف نمودن شرایط پسین با شکست مواجه می‌شود: *lock* با فراخوانی نخ، قفل نمی‌شود.

در بسیاری از موارد، پشته‌ی فراخوانی در پاسخ به فراخوانی صریح *std::terminate()* پیمایش نمی‌شود، و تنها در برخی موارد خاص، پیاده‌سازی مشخص می‌نماید که آیا عدم پیمایش پشته به‌وقوع خواهد پیوست و یا خیر.

نه به‌صورت ضمنی و نه به‌صورت صریح، توابع *std::abort()*، *std::quick_exit()* و *std::_Exit()* را فراخوانی نکنید. هنگامی‌که *terminate_handler* پیش‌فرض نصب می‌گردد و یا *terminate_handler* توسط فراخوانی *std::abort()* یا *std::_Exit()* پاسخ می‌دهد، نه به‌صورت ضمنی و نه به‌صورت صریح،

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه‌ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

*std::terminate()* را فراخوانی نکنید. خاتمه‌ی غیرعادی برنامه، دلیلی اصلی وقوع حملات انکار خدمات است.

البته، مجاز هستید یک تابع خاتمه (مانند *std::exit()*) را، که مخرب‌ها را به‌صورت امن اجرا و منابع را به‌درستی پاک می‌کند، فراخوانی کنید.

### ۱.۱.۸ نمونه‌کد ناسازگار

در این مثال، ممکن است فراخوانی *f()* (که توسط *std::exit()* به‌عنوان اداره‌کننده‌ی خروجی ثبت شده‌است)، منجر به فراخوانی *std::terminate()* شود. زیرا این امکان وجود دارد که *throwing_func()* استثنایی را ایجاد نماید.

```
#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
	}
	// ...
}
```

### ۲.۱.۸ راه‌حل سازگار

در این راه‌حل، *f()* با استفاده از *throwing_func()*، تمامی استثنائات ایجادشده را رسیدگی می‌کند و از ایجاد مجدد ممانعت به‌عمل می‌آورد.

```
#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
	}
}
```

مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه‌ای

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

### ۳.۱.۸ استثنائات

**ERR50-CPP-EX1**: می‌توانید پس از نمایش ماهیت برنامه به عملگر، صراحتا توابع *std::abort()*، *std::_Exit()* یا *std::terminate()* را، در شرایطی که خطای حیاتی رخ داده‌است و هیچ راهی برای بازیابی اطلاعات وجود ندارد، فراخوانی نمایید. مثال زیر، گواهی بر این مدعا است:

```
#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");
	}
}
```

می‌توان از ماکروی *assert()* تحت این استثنا استفاده کرد، زیرا اظهاراتی که با شکست مواجه می‌شوند، پیش از فراخوانی *std::abort()*، عملگر را از وجود خطایی استاندارد در پیاده‌سازی مطلع خواهند ساخت.

### ۴.۱.۸ ارزیابی خطر

تأمین شرایط خاتمه‌ی غیرعادی برنامه می‌تواند تبعات جبران‌ناپذیری از قبیل آزاد نشدن منابع، بسته شدن منابع، و موارد مشابه را به‌همراه داشته‌باشد. تمامی موارد مذکور، مقدمه‌ای برای وقوع حملات انکار خدمات هستند.

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه‌ای

| قانون | شدت/دقت | احتمال | هزینه‌ی اصلاح | اولویت | سطح |
|---|---|---|---|---|---|
| **ERR50-CPP** | پایین | محتمل | متوسط | P4 | L3 |

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

## ۲.۸ ERR51-CPP تمامی استثنائات را اداره کنید

وقتی استثنایی ایجاد می‌شود، کنترل به نزدیک‌ترین اداره‌کننده، که نوع آن با نوع استثنای ایجادشده انطباق دارد، واگذار می‌گردد. اگر مستقیما اداره‌کننده‌ی منطبقی برای بلاک *try* یافت نشود، جست‌وجوی پویای اداره‌کننده در سایر بلاک‌های *try* موجود همان نخ، ادامه می‌یابد.

اداره‌کننده‌ی خاتمه‌ی پیش‌فرض فراخوانده‌شده توسط *std::terminate()*، *std::abort()* را فرا می‌خواند. تابع *std::abort()* تابعی است که فرآیند را به‌صورت غیرعادی خاتمه می‌دهد. وقتی *std::abort()* فراخوانده می‌شود، و یا در صورتی‌که پیاده‌سازی، پشته را پیش از فراخوانی *std::terminate()* پوشش ندهد، ممکن است مخرب‌های اشیا فراخوانی نشوند و منابع، در وضعیت میانی باقی بمانند. خاتمه‌ی غیرعادی فرآیند، از دلایل اصلی وقوع حملات انکار خدمات به‌شمار می‌رود.

لازم است تمامی استثنائات توسط اداره‌کننده‌های هم‌نوع خود کنترل شوند. در چنین شرایطی، حتی اگر نتوان استثنائات را بازیابی نمود، استفاده از اداره‌کنندگان منطبق تضمین می‌کند که پشته به‌درستی پیمایش شده و منابع نیز، به‌خوبی جهت خاتمه‌ی فرآیند، مدیریت خواهند شد.

### ۱.۲.۸ نمونه‌کد ناسازگار

در این مثال، نه *f()* و نه *main()* استثنائات ایجادشده توسط *throwing_func()* را دریافت نمی‌کنند. به‌دلیل آن‌که اداره‌کننده‌ی منطبقی یافت نمی‌شود، *std::terminate()* فراخوانی می‌شود.

```
void throwing_func() noexcept(false);
void f()
{
        throwing_func();
}
int main()
{
        f();
}
```

### ۲.۲.۸ راه‌حل سازگار

در این راه‌حل، نقطه‌ی شروع اصلی، به تمامی استثنائات رسیدگی کرده و تضمین می‌کند که پشته توسط *main()* پیمایش می‌شود و نیز، امکان مدیریت بهینه‌ی منابع خارجی را فراهم می‌آورد.

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

```
void throwing_func() noexcept(false);
void f()
{
	throwing_func();
}
int main()
{
	try
	{
		f();
	}
	catch (...)
	{
		// Handle error
	}
}
```

### ۳.۲.۸ نمونه‌کد ناسازگار

در این مثال، نقطه‌ی ورودی تابع نخ *thread_start()*، استثنای ایجادشده توسط *throwing_func()* را دریافت نمی‌کند. اگر تابع نخ اولیه بنابر استثنای ایجادشده خارج شود، *std::terminate()* فراخوانی می‌شود.

```
#include <thread>
void throwing_func() noexcept(false);
void thread_start()
{
	throwing_func();
}
void f()
{
	std::thread t(thread_start);
	t.join();
}
```

### ۴.۲.۸ راه‌حل سازگار

در این راه‌حل، *thread_start()* تمامی استثنائات را رسیدگی می‌کند و از ایجاد مجدد، جلوگیری به‌عمل می‌آورد. بدین‌سان، این امکان را برای نخ فراهم می‌آورد که برنامه را به‌طور طبیعی خاتمه بخشد.

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

```
#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.join();
}
```

### ۵,۲,۸ ارزیابی خطر

تأمین شرایط خاتمه‌ی غیرعادی برنامه می‌تواند تبعات جبران‌ناپذیری از قبیل آزاد نشدن منابع، بسته شدن منابع، و موارد مشابه را به‌همراه داشته‌باشد. تمامی موارد مذکور، مقدمه‌ای برای وقوع حملات انکار خدمات هستند.

| قانون | شدت/دقت | احتمال | هزینه‌ی اصلاح | اولویت | سطح |
|---|---|---|---|---|---|
| ERR51-CPP | پایین | محتمل | متوسط | P4 | L3 |

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

## ۳,۸ ERR52-CPP از *setjmp()* یا *longjmp()* استفاده نکنید

می‌توان از امکانات کتابخانه‌ی استاندارد C، به نام‌های *setjmp()* و *longjmp()*، برای شبیه‌سازی ایجاد و دریافت استثنائات استفاده کرد. اگرچه، این امکانات مدیریت پویای منابع را نادیده می‌گیرند و منجر به رفتارهای تعریف‌نشده‌ای، از قبیل نشت منابع و حملات انکار خدمات، می‌شوند.

توابع فوق را فراخوانی نکنید. به جای استفاده از آنها می‌توانید از اصطلاحات *throw* و اعلانات *catch* بهره ببرید.

### ۱,۳,۸ نمونه کد ناسازگار

اگر یک اصطلاح *therow* سبب شود تا مخربی غیربدیهی فراخوانی گردد، فراخوانی *longjmp()* در چنین شرایطی، رفتار تعریف‌نشده‌ای را به‌همراه خواهد داشت. در مثال زیر، یک شی محلی به نام *Counter*، *longjmp()* را فراخوانی می‌کند. به‌دلیل آن‌که مخرب شی، غیربدیهی است، رفتار تعریف‌نشده‌ای به‌وقوع می‌پیوندد.

```
#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);
}
```

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

```
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;
}
```

### ۱,۱,۳,۸ جزئیات پیاده‌سازی

با اجرای کد فوق در Clang 3.8 در سیستم‌عامل لینوکس، نتایج زیر تولید می‌شود. این نتایج حاکی از آن است که وقتی *f()* خاتمه می‌یابد، برنامه قادر نیست متغیر محلی *Counter* را از بین ببرد، که مبین رفتاری تعریف‌نشده است.

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

### ۲,۳,۸ راه‌حل سازگار

این راه‌حل، از *throw* و *catch* به جای *setjmp()* و *longjmp()* استفاده می‌کند.

```
#include <iostream>
struct Counter
{
	static int instances;
	Counter()
	{
		++instances;
	}
	~Counter()
	{
		--instances;
	}
};
```

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

```
int Counter::instances = 0;
void f()
{
        Counter c;
        std::cout << "f(): Instances: "<< Counter::instances << std::endl;
        throw "Exception";
}
int main()
{
        try
        {
                f();
        }
        catch (const char *E)
        {
                std::cout << "From catch: Instances: "<< Counter::instances << std::endl;
        }
        std::cout << "After longjmp(): Instances: "<< Counter::instances << std::endl;
}
```

### ۱,۲,۳,۸ جزئیات پیاده‌سازی

خروجی کد فوق به‌صورت زیر خواهد بود.

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

### ۳,۳,۸ ارزیابی خطر

استفاده از *setjmp()* و *longjmp()*، به‌دلیل عدم تخریب صحیح منابع، منجر به حملات انکار خدمات خواهد شد.

| قانون | شدت/دقت | احتمال | هزینه‌ی اصلاح | اولویت | سطح |
|---|---|---|---|---|---|
| **ERR52-CPP** | پایین | محتمل | متوسط | P4 | L3 |

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

## ۴.۸ ERR53-CPP به کلاس‌های پایه و یا اعضای داده‌ای کلاس را در یک اداره‌کننده‌ی بلاک *try* سازنده یا مخرب ارجاع ندهید

وقتی اداره‌کننده‌ی بلاک *try* متعلق به یک سازنده، استثنایی را دریافت می‌کند، هر کلاس پایه‌ی ایجادشده و نیز اعضای کلاس شی، پیش از ورود اداره‌کننده، از بین می‌روند. به‌طور مشابه، زمانی‌که اداره‌کننده‌ی بلاک *try* متعلق به یک مخرب، استثنایی را دریافت می‌کند، تمامی کلاس‌های پایه‌ی و نیز اعضای نامتغیر کلاس شی، پیش از ورود اداره‌کننده، از بین می‌روند.

به‌همین دلیل، ارجاع به کلاس‌های پایه و اعضای کلاس در بلاک *try* سازنده یا مخرب، عقلانی نیست و رفتار تعریف‌نشده‌ای را به‌همراه خواهد داشت.

### ۱.۴.۸ نمونه‌کد ناسازگار

در این مثال، سازنده‌ی کلاس *C*، توسط یک بلاک *try* به تمامی استثنائات رسیدگی می‌کند. اگرچه این سازنده، با بازرسی فیلد عضو *str* خود، منجر به رفتار تعریف‌نشده‌ای می‌شود.

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

### ۲.۴.۸ راه‌حل سازگار

در این مثال، اداره‌کننده به جای بررسی پارامتر عضو داده‌ای کلاس، پارامتر سازنده را بررسی می‌نماید و از وقوع رفتار تعریف‌نشده ممانعت به‌عمل می‌آورد.

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

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

### ۳.۴.۸ ارزیابی خطر

دسترسی به داده‌ی غیرایستا در اداره‌کننده‌ی استثنای سازنده یا مخرب، می‌تواند منجر به رفتار تعریف‌نشده‌ای شود.

| قانون | شدت/دقت | احتمال | هزینه‌ی اصلاح | اولویت | سطح |
|---|---|---|---|---|---|
| ERR53-CPP | پایین | بعید | متوسط | P2 | L3 |

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

## ۵٫۸ ERR54-CPP اداره‌کنندگان *catch* باید انواع پارامتری خود را از بیشترین اشتقاقی تا کمترین اشتقاقی، مرتب کنند

اداره‌کنندگان بلاک *try* سعی دارند به‌ترتیب ظاهر شوند. در نتیجه، اگر دو اداره‌کننده، استثنایی را از یک کلاس پایه دریافت نمایند (مانند *std::exception*)، استثنایی که دارای بیشترین اشتقاقی است، اولویت دارد.

### ۱٫۵٫۸ نمونه‌کد ناسازگار

در این مثال، اولین اداره‌کننده تمامی استثنائات کلاس *B* و نیز کلاس *D* را دریافت می‌کند، زیرا کلاس *D* از نوع کلاس *B* است. در نتیجه، اداره‌کننده‌ی دوم هیچ استثنایی را دریافت نخواهد کرد.

```
// Classes used for exception handling
class B {};
class D : public B {};
void f()
{
	try
	{
		// ...
	}
	catch (B &b)
	{
		// ...
	}
	catch (D &d)
	{
		// ...
	}
}
```

### ۲٫۵٫۸ راه‌حل سازگار

در این راه‌حل، نخستین و دومین اداره‌کننده، به‌ترتیب، تمامی استثنائات کلاس‌های *D* و *B* را در اختیار می‌گیرند.

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

```
// Classes used for exception handling
class B {};
class D : public B {};
void f()
{
        try
        {
                // ...
        }
        catch (D &d)
        {
                // ...
        }
        catch (B &b)
        {
                // ...
        }
}
```

مدیریت مرکز امداد

### ۳.۵.۸ ارزیابی خطر

| قانون | شدت/دقت | احتمال | هزینه‌ی اصلاح | اولویت | سطح |
|---|---|---|---|---|---|
| **ERR54-CPP** | متوسط | احتمال زیاد | پایین | P18 | L1 |

هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

## ۶.۸ ERR55-CPP به مشخصات استثنا اولویت دهید

زمانی تابعی یک استثنا از نوع $E$ را مجاز می‌داند که عبارت ثابت[۵۷] در *مشخصات غیراستثنای* آن، *false* ارزیابی شود یا *مشخصات استثنای پویای*[۵۸] آن یک نوع $T$ را دربرگیرد، که اداره‌کننده‌ی نوع $T$، با استثنای نوع $E$ تطابق داشته‌باشد.

اگر تابعی استثنای دیگری غیر از *مشخصه‌ی استثنای* مجاز ایجاد کند، منجر به یک خاتمه‌ی تعریف‌شده در پیاده‌سازی خواهد شد.

اگر تابعی، که توسط یک *مشخصه‌ی استثنای پویا* فراخوانی شده‌است، استثنایی را از نوعی که با *مشخصه‌ی استثنا* تطابق ندارد ایجاد نماید، تابع *std::unexpected()* فراخوانی می‌شود. رفتار این تابع را می‌توان بازنویسی کرد، اما به‌صورت پیش‌فرض، منجر به استثنای *std::bad_exception* می‌شود. اگر *std::bad_exception* در *مشخصه‌ی استثنا* لیست شود، تابع *std::terminate()* فراخوانی خواهد شد.

به‌صورت مشابه، اگر تابعی که با *مشخصه‌ی غیر استثنا* اعلان شده‌است، استثنایی را ایجاد کند که منجر به *false* ارزیابی شدن *مشخصه‌ی غیر استثنا* شود، تابع *std::terminate()* فراخوانی خواهد شد.

فراخوانی *std::terminate()*، منجر به یک خاتمه‌ی تعریف‌شده در پیاده‌سازی برنامه می‌شود. به‌منظور جلوگیری از خاتمه‌ی ناهنجار برنامه، هر تابعی که یک *مشخصه‌ی استثنا* تعریف نماید، باید خود و هر تابعی را که فراخوانی می‌کند محدود سازد تا فقط استثناهای مجاز رخ دهند.

### ۱.۶.۸ نمونه‌کد ناسازگار

در این مثال، تابع دوم تنها منجر به *Exception1* می‌شود، اما ممکن *است Exception2* نیز رخ دهد.

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

[57] Constant-expression
[58] Dynamic-exception-specification

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

```
#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
}
```

### ۲.۶.۸ راه‌حل سازگار

این راه‌حل، استثناهای *foo()* را می‌گیرد.

```
#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
	}
}
```

### ۳.۶.۸ راه‌حل سازگار

این راه‌حل، یک *مشخصه‌ی استثنای پویا* برای *bar()* اعلان می‌کند، که تمام استثنائات حاصل از آن را پوشش می‌دهد.

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

```
#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();
}
```

### ۴,۶,۸ نمونه‌کد ناسازگار

در این مثال ، یک تابع به‌صورت عدم ایجاد استثنا اعلان می‌شود، اما هنگامی که حافظه‌ی مورد تقاضا نمی‌تواند تخصیص یابد، *std::vector::resize()* می‌تواند منجر به استثنا شود.

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

### ۵,۶,۸ راه‌حل سازگار

در این راه‌حل، *مشخصه‌ی غیر استثنای* تابع حذف می‌شود که در این حالت، تابع تمام استثنائات را مجاز می‌داند.

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

#### *۱,۵,۶,۸ جزئیات پیاده‌سازی*

برخی از عرضه‌کنندگان، تعمیم‌هایی را برای زبان و به‌منظور مشخص کردن این‌که آیا لازم است یک تابع استثنایی ایجاد نماید یا خیر، فراهم می‌کنند. به‌عنوان نمونه، مایکروسافت ویژوال استودیو، *__declspec((nothrow))* را فراهم نموده‌است و Clang از *__attribute__((nothrow))* پشتیبانی می‌کند. اکنون عرضه‌کنندگان، رفتار مشخص‌کننده‌ی تابعی که استثنا ایجاد نمی‌کند را با استفاده از این

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

استثنائات مستند نمی‌سازند. ایجاد استثنا از تابعی که با یکی از این تعمیم‌های زبان اعلان می‌گردد، به‌عنوان رفتار تعریف‌نشده در نظر گرفته می‌شود.

### ۶,۶,۸ ارزیابی خطر

ایجاد استثنائات غیرمنتظره، جریان کنترلی را مختل می‌کند و می‌تواند منجر به خاتمه‌ی نابه‌هنگام[59] و حمله‌ی انکار خدمات شود.

| قانون | شدت/دقت | احتمال | هزینه‌ی اصلاح | اولویت | سطح |
|---|---|---|---|---|---|
| ERR55-CPP | پایین | احتمال زیاد | پایین | P9 | L2 |

مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

[59] Premature

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکزماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

## ۷,۸ ERR56-CPP ایمنی استثنا را تضمین کنید

اداره نمودن مناسب خطاها و وضعیت‌های استثنایی برای ادامه‌ی عملکرد صحیح نرم‌افزار ضروری است. سازوکار ترجیحی برای گزارش خطاها در برنامه‌ی++C، استثنائات هستند نه کدهای خطا. تعدادی از تسهیلات زبان هسته‌ای[۶۰]، شامل *dynamic_cast*، *operator new()* و *typeid*، شکست‌ها را با کمک استثنائات گزارش می‌کنند. علاوه بر این، کتابخانه‌ی استاندارد ++C، از استثنائات برای گزارش چندین نوع مختلف از شکست‌ها استفاده می‌نماید. تعداد کمی از برنامه‌های ++C می‌توانند از برخی از این تسهیلات استفاده ننمایند. در نتیجه، اکثریت قریب به اتفاق برنامه‌های ++C باید برای وقوع استثنائات آماده باشند و هریک را به‌صورت مناسب اداره کنند.

از آنجاکه استثنائات، به یک برنامه مسیرهای کد معرفی می‌کنند، در نظر گرفتن اثرات دریافت چنین مسیری توسط کد و اجتناب از هر اثر نامطلوبی که ممکن است به‌وجود آید، امری پراهمیت به‌شمار می‌رود. برخی از چنین اثراتی شامل شکست در رهاسازی یک منبع اخذشده، یک نشت و نیز شکست در برقراری مجدد و یا نگهداری یک قسمت تغییرناپذیر کلاس[۶۱] پس از به روزرسانی جزیی یک شی را نتیجه خواهد داد. کدی که از چنین اثرات نامطلوبی اجتناب کند، ایمن از استثنا نامیده می‌شود.

بر مبنای اثرات ماقبل، جدول زیر سه نوع ضمانت ایمنی از استثنا را از بیشترین تا کمترین حالت مطلوب، مشخص می‌کند.

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

---

[۶۰] Core language

[61] Class invariant

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

| ضمانت | توصیف | مثال |
|---|---|---|
| **قوی** | تضمین ایمنی از استثنا قوی، یک ویژگی از یک عمل است، به‌طوری‌که، علاوه بر ارضای تضمین ایمنی از استثنا پایه، اگر عمل به‌خاطر استثنا خاتمه یافت، هیچ اثر قابل مشاهده‌ای روی وضعیت برنامه نگذارد. | ایمنی از استثنای قوی |
| **پایه** | تضمین ایمنی از استثنای پایه، یک ویژگی از یک عمل است، به‌طوری‌که اگر عمل به خاطر استثنا خاتمه یابد، نامتغیرهای وضعیت برنامه را حفظ می‌کند و جلوی نشت منابع را می‌گیرد. | ایمنی از استثنای پایه |
| **هیچکدام** | کدی که ایمنی از استثنا قوی و پایه را تضمین نمی‌کند. | بدون ایمنی از استثنا |

کدی که ایمنی از استثنای قوی را تضمین کند، ایمنی از استثنای پایه را نیز تضمین می‌نماید.

چون باید تمامی استثنائات در یک برنامه‌ی کاربردی اداره شوند، مطابق با ERR50-CPP، برنامه نباید به‌طور ناگهانی خاتمه یابد، و استثنائات به‌وجود آمده نباید برنامه را در یک وضعیت نامشخص (که در آن نامتغیرها نقض[۶۲] می‌شوند) رها کند. در واقع، برنامه باید ایمنی از استثنای پایه را برای تمام نامتغیرها فراهم کند و بتواند تصمیم بگیرد که برای برخی از نامتغیرها، ایمنی از استثنای قوی را فراهم نماید. در صورتی‌که اداره‌ی استثنا برای کنترل خاتمه‌ی برنامه یا بازیابی از یک وضعیت استثنایی استفاده شود، یک نامتغیر نقض‌شده برنامه را در وضعیتی رها می‌کند که احتمال دارد اجرای برازنده‌ی ادامه‌دار، آسیب‌پذیری‌های امنیتی ایجاد کند. از این‌رو، کدی که هیچ تضمینی برای ایمنی استثنا مهیا نمی‌کند، ناایمن است و باید آن را معیوب در نظر گرفت.

### ۱.۷.۸ نمونه‌کد ناسازگار (بدون ایمنی از استثنا)

مثال زیر، یک عملگر انتساب کپی ناقص[۶۳] را نشان می‌دهد. نامتغیرهای ضمنی کلاس، که اعضای *array* یک اشاره‌گر مجاز (و احتمالا تهی) هستند، و اعضای عنصر *nElems*، تعداد عناصر آرایه را، که توسط *array* به

[62] Violate
[63] Flawed

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

آن‌ها اشاره شده‌است، نشان می‌دهند. تابع، *array* را پس می‌گیرد و شمارنده‌ی عنصر را، پیش از تخصیص یک بلاک جدید از حافظه برای کپی، انتساب می‌دهد. در نتیجه، اگر عبارت *new* منجر به یک استثنا شود، تابع وضعیت هر دو متغیرهای عضو را به صورتی تغییر می‌دهد که نامتغیرهای ضمنی کلاس را نقض کند. متعاقباً، چنین شی‌ای در یک وضعیت نامشخص قرار می‌گیرد و هر عملی روی آن، شامل تخریب آن، منجر به یک رفتار تعریف‌نشده می‌شود.

```
#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;
		}
		// ...
};
```

مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

### ۲.۷.۸ راه‌حل سازگار (ایمنی استثنای قوی)

در این راه‌حل، عملگر انتساب کپی، ایمنی استثنا قوی را تضمین می‌کند. تابع فضای ذخیره‌سازی جدید را برای کپی، پیش از تغییر وضعیت شی، اختصاص می‌دهد. تنها پس از موفقیت تخصیص، تابع می‌تواند وضعیت شی

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

را تغییر دهد. علاوه بر این، با کپی کردن آرایه به فضایی که به تازگی تخصیص داده شده‌است (پیش از پس گرفتن آرایه‌ی موجود)، تابع از تست کردن خودانتسابی[۶۴] اجتناب می‌کند، که منجر به بهبود کارایی کد در موارد شایع می‌شود.

```
#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;
		}
		// ...
};
```

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

### ۳.۷.۸ ارزیابی خطر

کدی که ایمن از استثنا نیست، معمولاً منجر به نشت در منبع می‌شود که این امر باعث می‌گردد برنامه در یک وضعیت ناسازگار یا غیرمنتظره قرار گیرد. در نهایت، پس از ایجاد استثنا، منجر به یک رفتار تعریف‌نشده در برخی نقاط خواهد شد.

[64] Self-assignment

| قانون | شدت/دقت | احتمال | هزینه‌ی اصلاح | اولویت | سطح |
|---|---|---|---|---|---|
| **ERR56-CPP** | بالا | احتمال زیاد | بالا | P9 | L2 |

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

## ۸,۸ ERR57-CPP هنگام اداره‌ی استثنائات، منابع را نشت ندهید

بازپس‌گیری منابع هنگام بروز استثنائات امری مهم است. ممکن است یک استثنای ایجادشده منجر به نادیده گرفته شدن کد پاکسازی شود یا یک شی در یک وضعیت مقداردهی اولیه‌ی جزئی، رها شود. چنین شی‌ای می‌تواند ایمنی استثنای پایه را نقض کند (ERR56-CPP).

ایمنی از استثنا را تضمین کنید. بهتر است منابع به‌صورت خودکار و هنگامی‌که اشیا از محدوده خارج می‌شوند، با استفاده از الگوی طراحی RAII بازپس گرفته‌شوند. این روش، نیازی به نوشتن کد پاکسازی پیچیده در هنگام تخصیص منابع ندارد.

با این وجود، سازنده‌ها[65] چنین حفاظتی را ارائه نمی‌دهند. به‌دلیل آن‌که یک سازنده، درگیر تخصیص منابع است، در صورت خاتمه‌ی نابه‌هنگام، منابع تخصیص داده‌شده را به‌صورت خودکار آزاد نمی‌کند.

عموماً پیشنهاد می‌شود سازنده‌هایی که نمی‌توانند کارشان را تکمیل کنند، باید به جای خروج عادی و رهاسازی شی خود در یک وضعیت غیرکامل، استثنا ایجاد نمایند. نباید منابع حین ایجاد یک استثنا (مثلاً در حین ساخت یک شی) نَشت کنند.

### ۱,۸,۸ نمونه‌کد ناسازگار

در این کد، *pst*، هنگامی‌که *process_item* استثنایی ایجاد می‌کند، به‌صورت مناسب رها نمی‌شود و نشت منبع را به‌همراه خواهد داشت.

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

[65] Constructors

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

```
#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;
}
```

### ۲.۸.۸ راه‌حل سازگار (*delete*)

در این راه‌حل، اداره‌کننده‌ی استثنا، *pst* را با فراخوانی *delete* آزاد می‌سازد. با وجود این‌که این راه‌حل به‌صورت مناسب منابع خود را با استفاده از بندهای *catch* رها می‌کند، این رویه می‌تواند معایبی داشته‌باشد:

- هر پاکسازی مجزایی نیازمند بلاک‌های *try* و *catch* خود است.
- عمل پاکسازی نباید منجر به هیچ‌گونه استثنایی شود.

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

```
#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;
}
```

### ۳.۸.۸ راه‌حل سازگار (الگوی طراحی RAII)

روش بهتر این است که از RAII استفاده شود. این الگو هر شی را مجبور می‌سازد که در مواجهه با رفتار ناهنجار، خود را پاکسازی کند، و دیگر لازم نیست برنامه‌نویس این کار را انجام دهد. مزیت دیگر این روش آن است که نیازمند عباراتی برای اداره نمودن خطاهای تخصیص منابع نیست (مطابق با MEM52-CPP، تشخیص و اداره‌ی خطاهای تخصیص حافظه).

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

```
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.
```

### ۴,۸,۸ نمونه‌کد ناسازگار

در این مثال، ممکن است سازنده‌ی *C::C()* در تخصیص حافظه به *a* و *b* شکست بخورد، یا ممکن است در متد *init()*، استثنایی ایجاد کند. اگر *init()* استثنا ایجاد نماید، نه *a* و نه *b* آزاد نخواهند شد. به‌طور مشابه، اگر تخصیص *b* شکست بخورد، *a* آزاد نخواهد شد.

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

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

### ۵,۸,۸ راه‌حل سازگار (*try/catch*)

این راه‌حل، احتمال شکست ناشی از آزادسازی *a* و *b* را، در صورتی‌که استثنایی در حین تخصیص آنها و یا حین *init()* رخ دهد، کاهش می‌دهد.

```
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;
			}
		}
};
```

### ۶,۸,۸ راه‌حل سازگار (*std::unique_ptr*)

این راه‌حل، از *std::unique_ptr* برای ساخت اشیایی، که در صورت بروز مشکل در سازنده‌ی *C::C()*، خود را پاکسازی می‌کنند، استفاده شده‌است. *std::unique_ptr* قوانین RAII را به اشاره‌گرها اعمال می‌کند.

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکزماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

```
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();
                }
};
```

### ۷.۸.۸ ارزیابی خطر

در نهایت، نشت حافظه و سایر منابع منجر به خرابی برنامه خواهد شد. اگر یک مهاجم بتواند با وادار کردن ایجاد استثنا از طریق ارسال داده‌های دستکاری‌شده، منجر به نشت مکرر منابع شود، آنگاه مهاجم می‌تواند یک حمله‌ی انکار خدمت برقرار سازد.

| قانون | شدت/دقت | احتمال | هزینه‌ی اصلاح | اولویت | سطح |
|---|---|---|---|---|---|
| **ERR57-CPP** | پایین | محتمل | بالا | P2 | L3 |

مدیریت مرکز امداد و هماهنگی رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکزماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

## ۹.۸ ERR58-CPP پیش از آغاز اجرای *main()*، تمامی استثنائات را اداره نمایید

ذکر این نکته الزامی است که تمامی استثنائات می‌توانند، حتی در صورت استفاده‌ی دقیق از بلاک‌های *try*، گرفته شوند. وقتی شی‌ای را با دوره‌ی ذخیره‌سازی ایستا یا نخ فراخوانی می‌کنید، و آن شی در حوزه‌ی بلاک تابع فراخوانی نمی‌شود، باید سازنده‌ی نوع به‌صورت *noexcept* اعلان شود و با ERR55-CPP مطابقت داشته‌باشد. همچنین، مقداردهنده‌ی اولیه‌ی چنین اعلانی نباید یک استثنای دست‌نیافتنی ایجاد نماید. اگر چنین استثنایی پیش از اجرا و یا پس از اتمام اجرای *main()* ایجاد شود، فرصتی برای اداره‌کردن استثنا پیش نخواهد آمد و رفتار پیاده‌سازی‌محوری را نتیجه خواهد داد.

### ۱.۹.۸ نمونه‌کد ناسازگار

در این مثال، ممکن است سازنده‌ی *S* استثنایی ایجاد کند که وقتی *globalS* در طول شروع برنامه ساخته می‌شود، گرفته می‌شود.

```
struct S
{
        S() noexcept(false);
};
static S globalS;
```

### ۲.۹.۸ راه‌حل سازگار

این راه‌حل، *globalS* را به متغیری با دوره‌ی ذخیره‌سازی ایستا تبدیل کرده و امکان اخذ هر استثنای ایجادشده در طول ساخت شی را فراهم می‌سازد. زیرا، سازنده‌ی *S* نخستین‌بار، به جای این‌که حین آغاز برنامه اجرا شود، هنگامی‌که *globalS()* فراخوانی می‌شود اجرا خواهد شد. در این راه‌حل نیاز است برنامه‌نویس کد منبع را به‌گونه‌ای تغییر دهد که فراخوانی تابع *globalS()* جایگزین استفاده‌های پیشین *globalS* شود.

مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکزماهر
مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

```
struct S
{
	S() noexcept(false);
};
S &globalS()
{
	try
	{
		static S s;
		return s;
	}
	catch (...)
	{
		// Handle error, perhaps by logging it and gracefully terminating the application.
	}
	// Unreachable.
}
```

### ۳.۹.۸ نمونه‌کد ناسازگار

در این مثال، ممکن است سازنده‌ی *global* در طول آغاز برنامه، استثنایی ایجاد نماید. (سازنده‌ی *std::string* که یک مقدار * *const char* و یک شی تخصیص‌گر پیش‌فرض را می‌پذیرد، *noexcept* در نظر گرفته نمی‌شود و در نتیجه، تمامی استثنائات را مجاز می‌سازد). این استثنا توسط بلاک *try* تابع *main()* گرفته نمی‌شود. در نتیجه، *std::terminate()* فراخوانی می‌گردد و خاتمه‌ی نابه‌هنجار برنامه را در پی خواهد داشت.

```
#include <string>
static const std::string global("...");
int main()
try
{
	// ...
}
catch(...)
{
	// IMPORTANT: Will not catch exceptions thrown from the constructor of global
}
```

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

### ۴,۹,۸ راه‌حل سازگار

کد سازگار باید از فرار استثنائات در طول آغاز و خاتمه‌ی برنامه، ممانعت به عمل آورد. این راه‌حل از تعریف یک *std::string* در حوزه‌ی فضای نام عمومی جلوگیری می‌کند و در عوض، از یک *static char* * استفاده می‌نماید.

```
static const char *global = "...";
int main()
{
        // ...
}
```

### ۵,۹,۸ ارزیابی خطر

ایجاد استثنایی که گرفته نشود، منجر بهخاتمه‌ی نابه‌هنجار برنامه و در پی آن، حمله‌ی انکار خدمات خواهد شد.

| قانون | شدت/دقت | احتمال | هزینه‌ی اصلاح | اولویت | سطح |
|---|---|---|---|---|---|
| **ERR58-CPP** | پایین | احتمال زیاد | پایین | P9 | L2 |

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

## ۱۰,۸ ERR59-CPP استثنایی را خارج از مرزهای اجرا ایجاد نکنید

ایجاد یک استثنا به همکاری بین اجرای عبارت *throw* و عبور کنترل به یک وضعیت *catch* مناسب نیاز دارد. این همکاری به شکل منطقی، در زمان اجرا استفاده می‌شود و برای محاسبه‌ی اداره‌کننده‌ی درست استثنا، شکل می‌گیرد. برای کد تفسیرشده توسط یک مفسر ساده‌ی C++، جزئیات چگونگی ایجاد و گرفتن استثنائات می‌تواند به‌صورت امن نادیده گرفته شوند. با این‌وجود، هنگام ایجاد یک استثنا خارج از مرزهای اجرا باید اطمینان حاصل کرد که زمان اجرای منطقی استفاده‌شده میان دو سمت مرز اجرا، سازگار باشد.

مرز اجرایی محدوده‌ی بین کد تفسیرشده‌ی مفسرهای مختلف، متفاوت است. برای مثال، اعلان یک تابع می‌تواند در یک فایل سرآیند باشد اما تعریف آن در کتابخانه‌ای، که در زمان اجرا بارگذاری می‌شود، صورت گیرد. مرز اجرا، میان سایت فراخوانی در فایل اجرایی و پیاده‌سازی تابع در کتابخانه است. همچنین، برخی مرزها مرز ABI (رابط باینری برنامه‌ی کاربردی) نامیده می‌شوند، زیرا با قابلیت همکاری مرزهای برنامه کاربردی ارتباط دارند.

یک استثنا را زمانی خارج از مرز اجرا ایجاد کنید که هردو طرف مرز از یک ABI برای راه‌اندازی استثنا استفاده می‌کنند.

### ۱,۱۰,۸ نمونه کد ناسازگار

در این مثال، یک استثنا از یک تابع کتابخانه ای بمنظور سیگنال به خطا ایجاد می‌شود. باوجود اینکه استثنا از نوع اسکالر است، این کد می‌تواند همچنان منجر به اجرای غیرطبیعی برنامه شود اگر کتابخانه و برنامه به ABIهای متفاوت چسبیده باشند.

مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

```
// 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
	}
}
```

### ۲,۱۰,۸ راه‌حل سازگار

در این راه‌حل، خطای تابع کتابخانه‌ای، به جای یک استثنا توسط یک مقدار بازگشتی نشان داده می‌شود.

```
// library.h
void func() noexcept(false); // Implemented by the library
// library.cpp
int func() noexcept(true)
{
	// ...
	if (/* ... */)
	{
		return 42;
	}
	// ...
	return 0;
}
// application.cpp
#include "library.h"
```

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

```
void f()
{
        if (int err = func())
        {
                // Handle error
        }
}
```

### ۳.۱۰.۸ ارزیابی خطر

اثرات ناشی از ایجاد یک استثنا خارج از مرزهای اجرایی، به جزییات پیاده‌سازی مکانیک‌های اداره‌ی استثنا واسته است، که این بازه می‌تواند از رفتار درست تا تعریف‌نشده، متغیر باشد.

| قانون | شدت/دقت | احتمال | هزینه‌ی اصلاح | اولویت | سطح |
|---|---|---|---|---|---|
| ERR59-CPP | بالا | محتمل | متوسط | P12 | L1 |

مدیریت مرکز هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکزماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

## ۱۱,۸ ERR60-CPP اشیای استثنا نباید از طریق کپی قابل ساخت، ایجاد شوند

هنگام ایجاد یک استثنا، عملوند شی استثنا، از عبارت *throw* به یک شی موقتی کپی می‌شود که آن شی برای مقداردهی اولیه اداره‌کننده استفاده می‌گردد.

اگر کپی تابع سازنده‌ی نوع شی استثنا در طول کپی مقداردهی اولیه ایجاد گردد، *std::terminate()* فراخوانی خواهد شد، که رفتار نامشخصی را با خود به‌همراه خواهد شد.

باید کپی تابع سازنده‌ی شی‌ای، که به‌عنوان یک استثنا ایجاد شده‌است، به‌صورت *noexcept* اعلان گردد تا شامل هرگونه کپی تابع سازنده‌ی تعریف‌شده‌ی ضمنی شود.

استاندارد C++ به تابع کپی سازنده اجازه می‌دهد هنگام مقداردهی اولیه‌ی شی استثنا، حذف شود تا در صورتی‌که استثنای موقتی ایجاد می‌گردد، مقداردهی اولیه انجام پذیرد. شمار متعددی از پیاده‌سازی‌های مفسر مدرن از هر دو روش بهینه‌سازی استفاده می‌کنند. با این‌وجود، همچنان تابع کپی سازنده‌ی یک شی استثنا نباید یک استثنا را ایجاد کند، زیرا نیاز نیست تا مفسرها فراخوانی تابع کپی سازنده را در تمامی وضعیت‌ها حذف نمایند. از طرفی، پیاده‌سازی مشترک *std::exception_ptr*، حتی اگر از یک عبارت *throw* حذف شود، یک تابع کپی سازنده را فراخوانی خواهد کرد.

### ۱,۱۱,۸ نمونه‌کد ناسازگار

در این مثال، *f()* استثنایی از نوع *S* ایجاد می‌کند. با این‌وجود، به‌دلیل آن‌که *S* دارای یک عضو داده‌ای *std::string*/ست و تابع کپی سازنده‌ی *std::string* به‌عنوان *noexcept* اعلان نشده‌است، تابع کپی سازنده‌ی تعریف‌شده‌ی ضمنی *S* نیز به‌عنوان *noexcept* اعلان نخواهد شد. دروضعیت‌های کمبود حافظه، ممکن است تابع کپی سازنده‌ی *std::string* قادر به معرفی حافظه‌ی کافی برای تکمیل عمل کپی نباشد، که منجر به ایجاد استثنای *std::bad_alloc* خواهد شد.

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکزماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

```
#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
	}
}
```

### ۲,۱۱,۸ راه‌حل سازگار

در این راه‌حل فرض بر آن است که نوعی از شی استثنا می‌تواند از *std::runtime_error* ارث ببرد، یا این‌که قادر است به‌صورت مستقیم استفاده شود. برخلاف *std::string*، یک *std::runtime_error* نیاز دارد پیغام خطایی با طول دلخواه را به‌درستی اداره کند تا استثنا امن گردد و تضمین نماید که تابع کپی سازنده، ایجاد نخواهد شد.

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکزماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

```
#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
	}
}
```

### ۳.۱۱.۸ راه‌حل سازگار

در این راه‌حل، اگر نوع استثنا نتواند برای ارث‌بری از *std::runtime_error* تعریف و اصلاح شود، یک عضو داده‌ای آن نوع، یک استراتژی پیاده‌سازی مشروع خواهد بود.

```
#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();
		}
};
```

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

```
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
	}
}
```

### ۴,۱۱,۸ارزیابی خطر

اگر برنامه‌ی کاربردی به‌صورت ناهنجار پایان یابد می‌تواند منجر شود که منابع آزاد نگردند، بسته شوند، و یا مواردی از این دست روی دهد. پیامد ناشی از چنین رفتاری، حملات انکار خدمات خواهد بود.

| قانون | شدت/دقت | احتمال | هزینه‌ی اصلاح | اولویت | سطح |
|---|---|---|---|---|---|
| **ERR60-CPP** | پایین | محتمل | متوسط | P4 | L3 |

مدیریت مرکز امداد و هماهنگی رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

## ۱۲,۸ ERR61-CPP استثناها را به‌وسیله‌ی ارجاع مقدار سمت چپ بگیرید

هنگامی‌که استثنایی ایجاد می‌شود، از مقدار شی موجود در عبارت ایجادشده، برای مقداردهی اولیه‌ی شی موقت ناشناس، که exception object نام دارد، استفاده می‌شود. این نوع شی استثنا برای انتقال کنترل به نزدیکترین اداره‌کننده‌ی گرفتن[۶۶]، استفاده می‌شود که شامل یک اعلان استثنا به‌همراه یک نوع هماهنگ با آن است.

به‌دلیل آن‌که متغیر اعلان‌شده به‌وسیله‌ی *exception-declaration* از طریق کپی مقداردهی اولیه شده‌است، ممکن است تکه کردن شی استثنا به‌عنوان بخشی از عمل کپی، منجر به ازدست رفتن اطلاعات استثنای باارزش و بازیابی نادرست خطا شود. همچنین، اگر تابع کپی سازنده‌ی شی استثنا، استثنایی را ایجاد نماید، مقداردهی اولیه‌ی کپی‌شده از شی *exception-declaration* منجر به رفتار نامشخص خواهد شد. همواره، استثنائات را به‌وسیلهی ارجاع مقدار سمت چپ بگیرید، مگر آن‌که نوع آن،از نوع بدیهی باشد.

### ۱,۱۲,۸ نمونه‌کد ناسازگار

دراین مثال، یک شی از نوع *S* برای مقداردهی اولیه‌ی شی استثنا استفاده می‌گردد که بعدا، با یک *exception-declaration* از نوع *std::exception* گرفته می‌شود. *exception-declaration* منطبق بر نوع شی استثنا است، بنابراین، متغیر *E* توسط شی استثنا، از طریق کپی مقداردهی اولیه می‌شود که منجر به تکه شدن شی استثنا خواهد شد. درنتیجه، خروجی این کد به جای "My custom exception"، مقدار تعریف‌شده‌ی پیاده‌سازی بازگشتی از فراخوانی *std::exception::what()* خواهد بود.

```
#include <exception>
#include <iostream>
struct S : std::exception
{
	const char *what() const noexcept override
	{
		return "My custom exception";
	}
};
```

[۶۶] Catch handler

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکزماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

```
void f()
{
	try
	{
		throw S();
	}
	catch (std::exception e)
	{
		std::cout << e.what() << std::endl;
	}
}
```

### ۲,۱۲,۸ راه‌حل سازگار

در این راه‌حل، متغیر تعریف‌شده توسط *exception-declaration*، یک ارجاع مقدار سمت چپ است. فراخوانی *what()*، به جای اجرای *std::exception::what()*، منجر به اجرای *S::what()* خواهد شد.

```
#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;
	}
}
```

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

### ۳.۱۲.۸ارزیابی خطر

تکه شدن شی منجر به ناهنجاری در اجرای برنامه می‌شود. به‌طور کلی، این موضوع مشکلی برای استثنا به‌شمار نمی‌آید، اما میت‌واند منجر به رفتار غیرقابل انتظاری شود که به فرضیات ایجادشده توسط اداره‌کننده‌ی استثنا وابسته‌است.

| قانون | شدت/دقت | احتمال | هزینه‌ی اصلاح | اولویت | سطح |
|---|---|---|---|---|---|
| ERR61-CPP | پایین | بعید | پایین | P3 | L3 |

مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

## ۱۳,۸ ERR62-CPP خطاها را هنگام تبدیل یک رشته به یک عدد تشتخیص دهید

فرآیند تجزیه‌ی[۶۷] یک عدد صحیح یا اعشاری از یک رشته می‌تواند تعدادی خطا تولید کند؛ ممکن است رشته شامل عدد نباشد؛ ممکن است شامل یک عدد خارج از بازه باشد (مانند یک عدد صحیح بزرگتر از *INT_MAX*)؛ همچنین می‌تواند شامل اطلاعات اضافی پس از عدد باشد، که می‌تواند بعد از تبدیل استفاده شوند و یا بلااستفاده باقی بمانند. ممکن است شرایط خطایی مذکور، هنگامی‌که یک تبدیل رشته به عدد انجام می‌گیرد، با استفاده از یک جریان ورودی فرمت‌شده (مانند *std::istream* یا رویه‌ی محلی *num_get<>*)، تشخیص داده‌شده و برطرف شوند.

هنگام فراخوانی یک تابع جریان ورودی فرمت‌شده، مانند *istream::operator>>()*، اطلاعات مرتبط با خطاهای تبدیل، از طریق توابع عضو ارث‌بری *basic_ios::good()*، *basic_ios::bad()* و *basic_ios::fail()* یا از طریق اداره‌کننده‌ی استثنا (در صورتی‌که بر جریان، فعال شده‌باشد)، مورد پرس‌وجو قرار خواهد گرفت.

هنگام فراخوانی *num_get<>::get()*، اطلاعات مربوط به خطاهای تبدیل، از طریق پارامتر *ios_base::iostate&* به فراخواننده برگشت داده می‌شوند. همواره، وضعیت خطای تبدیل رشته به مقدار عددی را به‌وضوح بررسی نمایید.

### ۱,۱۳,۸ نمونه‌کد ناسازگار

در این مثال، مقدار عددی چندگانه، از جریان ورودی استاندارد تبدیل می‌شود. با این‌وجود، اگر متن دریافت‌شده از جریان ورودی استاندارد نتواند به یک مقدار عددی (که قابل نمایش توسط *int* است) تبدیل شود، مقادیر حاصل ذخیره‌شده در متغیرهای *i* و *j*، مقادیر دور از انتظاری خواهند بود.

```
#include <iostream>
void f()
{
        int i, j;
        std::cin >> i >> j;
        // ...
}
```

[۶۷] Parsing

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکزماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

برای مثال اگر متن 12345678901234567890، مقدار تبدیل شده‌ی نخست خوانده‌شده از جریان ورودی استاندارد باشد، مقدار متغیر *i*، مقدار *std::numeric_limits<int>::max()* خواهد بود و متغیر *j*، مقداردهی اولیه نخواهد شد. اگر متن *abcdefg*، مقدار تبدیل شده‌ی اول خوانده‌شده از جریان ورودی استاندارد باشد، سپس متغیر *i* مقدار ۰ را اتخاذ خواهد کرد و متغیر *j*، مقداردهی اولیه نخواهد شد.

### ۲.۱۳.۸ راه‌حل سازگار

در این راه‌حل، استثنا در حالت فعال قرار دارد، به‌گونه‌ای که هیچ شکستی در تبدیل منجر به ایجاد استثنا نمی‌شود. با این‌وجود، این روش نمی‌تواند بین مقادیر معتبر و نامعتبر تمییز قائل شود و باید تمامی مقادیر، نامعتبر فرض شوند. هر دو پرچم *badbit* و *failbit* جهت حصول اطمینان از این‌که خطاهای تبدیل و نیز فقدان جامعیت با جریان، به‌عنوان استثنا در نظر گرفته‌شوند، تنظیم می‌گردند.

```
#include <iostream>
void f()
{
	int i, j;
	std::cin.exceptions(std::istream::failbit | std::istream::badbit);
	try
	{
		std::cin >> i >> j;
		// ...
	}
	catch (std::istream::failure &E)
	{
		// Handle error
	}
}
```

### ۳.۱۳.۸ راه‌حل سازگار

در این راه‌حل، هرمقدار تبدیل‌شده‌ی خوانده‌شده از جریان ورودی استاندارد، پیش از خوانده شدن مقدار بعدی، به‌منظور تایید اعتبار، تست می‌شود و امکان بازیابی پایه‌ی مقدار پیشین را فراهم می‌آورد. ابتدا، *std::istream::fail()* را چک می‌کند تا بفهمد آیا بیت شکست به‌منظور شکست تبدیل، تنظیم شد و یا آیا بیت بد به‌منظور از دست رفتن تجمیع با شی جریان، تنظیم گردید یا خیر. مبرهن است که درصورت

مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

مواجهه با شرایط شکست، جریان ورودی و سپس کاراکترها خوانده می‌شوند. همچنین، کاراکتر فضای خالی[۶۸] (' ') دورانداخته می‌شود. تصحیح خطا در این مثال تنها زمانی کار می‌کند که کاراکتر فضای خالی (که حدود بین دو مقدار عددی را تعیین می‌کند) تبدیل شود.

```
#include <iostream>
#include <limits>
void f()
{
	int i;
	std::cin >> i;
	if (std::cin.fail())
	{
		// Handle failure to convert the value.
		std::cin.clear();
		std::cin.ignore( std::numeric_limits<std::streamsize>::max(), ' ');
	}
	int j;
	std::cin >> j;
	if (std::cin.fail())
	{
		std::cin.clear();
		std::cin.ignore(std::numeric_limits<std::streamsize>::max(), ' ');
	}
	// ...
}
```

### ۴,۱۳,۸ارزیابی خطر

آسیب‌پذیری ناشی از نقض این قانون به‌ندرت اتفاق می‌افتد، مگر آن‌که در یک کد حساس به امنیت روی دهد. با این‌وجود، نقض این قانون می‌تواند به آسانی منجر به از دست دادن یا عدم تفسیر داده شود.

| قانون | شدت/دقت | احتمال | هزینه‌ی اصلاح | اولویت | سطح |
|---|---|---|---|---|---|
| ERR62-CPP | متوسط | بعید | متوسط | P4 | L3 |

[۶۸] Space

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران

مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

# فصل نهم

# برنامه‌نویسی شی‌گرا (OOP)

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکزماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

## ۱,۹ OOP50-CPP توابع مجازی را از طریق سازنده‌ها یا مخرب‌ها فراخوانی نکنید

توابع مجازی امکان ارسال ایستای فراخوانی تابع عضو را در زمان اجرا و بر اساس نوع شی پویایی، که تابع عضو از آن فراخوانی شده‌است، فراهم می‌آورند. این قرارداد از روند برنامه‌نویسی شی‌گرایی، که به‌طور معمول با وراثت و دوباره‌نویسی توابع همراه است، پشتیبانی می‌کند. تابع موردنظر، زمانی فراخوانی می‌شود که یک تابع عضو غیرمجازی فراخوانی گردد و یا از یک عبارت دسترسی به عضوی از کلاس جهت معنی کردن یک فراخوانی استفاده می‌شود. در غیر این‌صورت، یک تابع مجازی برای ملغی‌گر نهایی در نوع پویای عبارت شی فراخوانی می‌شود. اگرچه، درهنگام ایجاد یا تخریب یک شی، قوانین ارسال روش مجازی به آن شی محدود است.

یک تابع مجازی را، که سعی می‌کند شی تحت ساخت یا تخریب را فراخوانی نماید، به‌صورت مستقیم یا غیرمستقیم از یک سازنده یا مخرب فراخوانی نکنید. زیرا، ترتیب سازنده‌ها از کلاس‌های اصلی آغاز می‌گردد و به کلاس‌های اشتقاقی‌تر انتقال می‌یابد، و تلاش برای فراخوانی یک تابع کلاس اشتقاقی از یک کلاس مبنای ساخت، خطرناک است. کلاس اشتقاقی این شانس را نداشته‌است که منابع خود را مقداردهی اولیه کند. به‌همین دلیل است که فراخوانی یک تابع مجازی از یک سازنده به یک فراخوانی تابع در یک کلاس اشتقاقی‌تر منجر نمی‌شود. به‌طور مشابه، یک شی، عکس ترتیب ساخت، تخریب می‌شود. در نتیجه، ممکن است تلاش برای فراخوانی یک تابع در یک کلاس اشتقاقی‌تر و از طریق یک مخرب، باعث دسترسی به منابعی شود که قبلا آزاد شده‌اند.

### ۱,۱,۹ نمونه‌کد ناسازگار

دراین مثال، کلاس مبنا سعی دارد منابع یک شی را از طریق فراخوانی توابع مجازی سازنده و مخرب، در اختیار بگیرد و آزاد کند. اگرچه، سازنده‌ی *B::B()*، به جای *D::seize()*، *B::seize()* و مخرب *B::~B()*، به جای *D::release()*، *B::release()* را فراخوانی می‌کنند.

مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکزماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

```
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();
                }
};
```

در نتیجه‌ی اجرای این کد، هیچ یک از منابع کلاس اشتقاقی، در حین مقداردهی اولیه و تخریب شی، از نوع *D* گرفته یا آزاد نمی‌شوند. سازنده‌ی *D*، به هنگام فراخوانی *seize()* از *B::B()* وارد نگردیده‌است. در نتیجه، رفتار شی در دست ساخت، به جای فراخوانی *D::seize()*، منجر به فراخوانی *B::seize()* می‌شود. شرایط مشابهی برای فراخوانی *release()* در مخرب کلاس اصلی پیش می‌آید. اگر توابع *seize()* و *release()* به‌صورت توابع مجازی صرف تعریف شوند، رفتار تعریف‌نشده‌ای روی خواهد داد.

### ۲.۱.۹ راه‌حل سازگار

در راه‌حل سازگار، سازنده‌ها و مخرب‌ها، یک عضو تابع غیر مجازی و شخصی (با پسوند *mine*) را به جای یک تابع مجازی فراخوانی می‌کنند. در نتیجه، هر کلاس، مسئول گرفتن و آزادسازی منابع خود است.

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

```
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();
                }
};
```

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

### ۳.۱.۹ استثنائات

**OOP50-CPP-EX1**: به‌دلیل آن‌که موارد استفاده‌ی معتبری وجود دارند، که شامل فراخوانی توابع مجازی ناخالص از سازنده‌ی یک کلاس هستند، فراخوانی تابع مجازی با یک ID واجد الشرایط صریح ممکن است. ID واجدالشرایط به نگهدارنده‌های کد نشان می‌دهد که رفتار مورد انتظار، متعلق به کلاسی است که تحت ساخت یا تخریب بوده تا ملغی‌گر نهایی برای فراخوانی تابع باشد.

```
struct A
{
	A()
	{
		// f(); // WRONG!
		A::f(); // Okay
	}
	virtual void f();
};
```

این امکان وجود دارد که یک تابع مجازی حاوی *final virt-specifier* را از سازنده یا مخرب فراخوانی کنیم. به‌عنوان مثال:

```
struct A
{
	A();
	virtual void f();
};
struct B : A
{
	B() : A()
	{
		f(); // Okay
	}
	void f() override final;
};
```

به‌طور مشابه، امکان فراخوانی یک تابع مجازی از طریق سازنده یا مخرب یک کلاس، که حاوی *final class-virt-specifier* است، وجود دارد. به‌عنوان مثال:

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکزماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

```
struct A
{
        A();
        virtual void f();
};
struct B final : A
{
        B() : A()
        {
                f(); // Okay
        }
        void f() override final;
};
```

در هردو مثال، لازم است f() ملغی‌گر نهایی باشد تا رفتار ثابت تابع را به هنگام فراخوانی تضمین نماید.

### ۴,۱,۹ ارزیابی خطر

| قانون | شدت/دقت | احتمال | هزینه‌ی اصلاح | اولویت | سطح |
|---|---|---|---|---|---|
| OOP50-CPP | پایین | بعید | متوسط | P2 | L3 |

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

## ۲,۹ OOP51-CPP اشیای اشتقاقی را نبُرید

معمولا، شی اشتقاقی از یک کلاس پایه، شامل متغیرهای عضو اضافی است که کلاس پایه را گسترش می‌دهند. هنگام تخصیص مقدار یا کپی شی از نوع اشتقاقی به یک شی نوع پایه، متغیرهای عضو اضافی مذکور کپی نمی‌شوند، زیرا کلاس پایه فضای کافی برای ذخیره‌ی آنها ندارد. این عمل، بریدن اشیا نام دارد، زیرا اشیای اضافی از تکه تکه کردن شی حاصل به‌دست می‌آیند.

یک شی پایه را با یک شی اشتقاقی مقداردهی اولیه نکنید، مگر از طریق مراجع، اشاره‌گرها، موارد انتزاعی شبه اشاره‌گر (مانند *std::unique_ptr* یا *std::shared_ptr*).

### ۱,۲,۹ نمونه‌کد ناسازگار

در این نمونه‌کد، یک شی اشتقاقی *Manager* حاوی مقدار، از تابعی که شی پایه‌ی *Employee* را می‌پذیرد، گذر می‌کند. در نتیجه، اشیای *Manager* تکه تکه می‌شوند، که در صورت فراخوانی تابع *print()*، منجر به از دست رفتن اطلاعات یا بروز یک رفتار غیرمنتظره خواهد شد.

```
#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;
```

مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

```
        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);
}
```

وقتی *f ()* به‌همراه پارامتر *designer* فراخوانی می‌شود، پارامتر رسمی در *f()* تکه تکه شده و اطلاعات از دست می‌رود. وقتی شی *e* چاپ می‌گردد، *Employee::print()* به جای *Manager::print()* فراخوانی می‌شود و خروجی زیر را نتیجه می‌دهد:

```
Employee: Jane Doe
```

### ۲.۲.۹ راه‌حل سازگار (اشاره‌گرها)

با استفاده از همان تعاریف کلاس در نمونه‌کد ناسازگار، این راه‌حل، تعریف *f()* را به‌گونه‌ای تغییر می‌دهد تا به اشاره‌گرهای خام شی نیاز داشته باشد و بتواند مشکل تکه کردن را حل نماید.

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

```
// Remainder of code unchanged...
void f(const Employee *e)
{
	if (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);
}
```

اشاره‌گرهای تهی را به هنگام اجرای *f()* مجددا ارجاع ندهید. با این توصیف، برنامه به‌درستی خروجی زیر را نتیجه می‌دهد:

```
Employee: Joe Smith
Employee: Bill Jones
Manager: Jane Doe
Assistant:
	Employee: Bill Jones
```

### ۳.۲.۹ راه‌حل سازگار (مراجع)

راه‌حل سازگار اثبات‌شده، که نیازی به محافظت در برابر اشاره‌گرهای تهی در *f()* ندارد، به جای اشاره‌گرها از مراجع استفاده می‌کند.

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

```
// ... 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);
}
```

### ۴,۲,۹ راه‌حل سازگار (غیر قابل نسخه‌برداری)

هر دو راه‌حل سازگار پیشین، برای این‌که با روشی سازگار و با استفاده‌ی موردنظر از وراثت کلاسی اعلان شوند، به مشتریان نوع *Employee* و *Manager* وابسته هستند. راه‌حل ارائه‌شده در این بخش، تضمین می‌نماید که مشتریان قادر نیستند تصادفا اشیا را، با حذف قابلیت کپی در زمان مقداردهی اولیه‌ی یک شی اشتقاقی از *Noncopyable*، تکه تکه کنند. اگر مبادرت به مقداردهی اولیه مبتنی بر کپی شود، مانند مورد مشابه در تعریف اصلی *f()*، برنامه بدفرم خواهد شد و یک تشخیص صورت خواهد گرفت. اگرچه، چنین راه‌حلی، شی *Manager* را از تلاش برای مقداردهی مبتنی بر کپی شی *Employee* خود باز می‌دارد، و بسیار ماهرانه، معناگرایی کلاس وراثت را تغییر می‌دهد.

```
#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;
		}
```

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

```
        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), assistant(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);
}
```

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

### ۵,۲,۹ نمونه‌کد ناسازگار

این نمونه‌کد، از همان تعاریف کلاس *Employee* و *Manager* مثال پیش استفاده می‌کند و تلاش دارد شی *Employee* را در یک *std::vector* ذخیره کند. اگرچه، چون *std::vector* به لیستی از عناصر نیاز دارد، تکه کردن اتفاق می‌افتد.

```
#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);
};
```

### ۶,۲,۹ راه‌حل سازگار

این راه‌حل، برای برطرف نمودن مشکل تکه‌سازی، از برداری از اشیای *std::unique_ptr* استفاده می‌کند.

```
#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;
	}
}
```

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه‌ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

```
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);
}
```

### ۷.۲.۹ ارزیابی خطر

تکه شدن باعث از بین رفتن اطلاعات می‌شود و ممکن است متعاقبا، اجرای غیرطبیعی و یا حمله‌ی انکار خدمات را به‌همراه داشته‌باشد.

| قانون | شدت/دقت | احتمال | هزینه‌ی اصلاح | اولویت | سطح |
|---|---|---|---|---|---|
| OOP51-CPP | پایین | محتمل | متوسط | P4 | L3 |

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکزماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

## ۳,۹ OOP52-CPP یک شی چندریختی را بدون مخرب مجازی حذف نکنید

شی نوع کلاس اشتقاقی را از طریق یک اشاره‌گر به کلاس پایه‌ی آن، که حاوی یک مخرب غیرمجازی است، حذف نکنید. در عوض، لازم است یک مخرب مجازی از کلاس پایه دفاع کند. در واقع، حذف یک شی از طریق اشاره‌گر به یک نوع، بدون استفاده از یک مخرب غیرمجازی، رفتار تعریف‌نشده‌ای را با خود به‌همراه خواهد داشت.

### ۱,۳,۹ نمونه‌کد ناسازگار

دراین نمونه‌کد، *b* یک اشاره‌گر چندریختی است که نوع ایستا و پویای آن به‌ترتیب، *Base* * و *Derived* * هستند. درصورت حذف b، شاهد یک رفتار تعریف‌نشده خواهیم بود، زیرا *Base* فاقد مخرب مجازی است. یک مخرب که به‌صورت صریح تعریف شده‌است، یک عضو درون‌خطی عمومی آن کلاس به‌شمار می‌رود و حتی در حضور سایر توابع مجازی، به‌صورت مجازی تعریف نمی‌شود.

```
struct Base
{
        virtual void f();
};
struct Derived : Base {};
void f()
{
        Base *b = new Derived();
        // ...
        delete b;
}
```

### ۲,۳,۹ نمونه‌کد ناسازگار

دراین نمونه‌کد، عملیات‌های یک شی اشاره‌گر هوشمند جایگزین اشاره‌گر صریح شده‌است، که نشان می‌دهد اشاره‌گر هوشمند نیز مانند سایر اشاره‌گرها از مشکل مشابه رنج می‌برد. چون حذف‌کننده‌ی پیش‌فرض *std::unique*_ptr، delete را بر مقدار داخلی اشاره‌گر فراخوانی می‌کند، رفتاری مشابه نمونه‌کد ناسازگار پیشین مشاهده خواهد شد.

مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه‌ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکزماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

```
#include <memory>
struct Base
{
	virtual void f();
};
struct Derived : Base {};
void f()
{
	std::unique_ptr<Base> b = std::make_unique<Derived()>();
}
```

### ۳,۳,۹ راه‌حل سازگار

در این راه‌حل، مخرب متعلق به *Base*، دارای مخربی مجازی است که به‌صورت صریح تعریف شده‌است و اطمینان می‌دهد عملیات حذف چندریختی، منجر به رفتاری تعریف‌شده خواهد شد.

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

### ۴,۳,۹ ارزیابی خطر

تلاش برای حذف یک شی چندریختی، که دارای مخربی است که به‌صورت مجازی تعریف‌نشده است، رفتار تعریف‌نشده‌ای را نتیجه می‌دهد، که در عمل عواقب بالقوه‌ای همچون خاتمه‌ی غیرطبیعی برنامه و نیز نشت حافظه را به دنبال دارد.

| قانون | شدت/دقت | احتمال | هزینه‌ی اصلاح | اولویت | سطح |
|---|---|---|---|---|---|
| OOP52-CPP | پایین | احتمال زیاد | پایین | P9 | L2 |

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

## ۴.۹ OOP53-CPP مقداردهنده‌های اولیه‌ی اعضای سازنده را به‌ترتیب استاندارد بنویسید

لیست مقدار دهنده‌ی اولیه‌ی عضو یک کلاس سازنده، این امکان را فراهم می‌آورد تا به اعضا، مقادیر اولیه‌ی خاصی تخصیص داده شود و سازنده‌های کلاس پایه، با پارامترهای خاصی فراخوانی شوند. اگرچه، ترتیبی که مقداردهی اولیه انجام می‌گیرد ثابت است و به ترتیبی که در لیست مقداردهنده‌ی اولیه عضو نوشته شده‌است، بستگی ندارد.

در نتیجه، ترتیبی که مقداردهنده‌های اولیه در آغازگر عضو ظاهر می‌شوند، نامرتبط است. ترتیبی که اعضا مقداردهی اولیه می‌شوند، شامل مقداردهی کلاس پایه، با ترتیب اعلان متغیرهای کلاس یا لیست تعیین‌کننده‌ی کلاس پایه مشخص می‌گردند. نوشتن مقداردهنده‌های اولیه، با ترتیبی غیر از ترتیب استاندارد، ممکن است باعث بروز رفتار تعریف‌نشده، مانند خواندن حافظه‌ی مقداردهی اولیه‌نشده، گردد.

همیشه، مقداردهنده‌های اولیه‌ی عضو را به‌ترتیب استاندارد در یک سازنده بنویسید: نخست، کلاس‌های مستقیم پایه، به ترتیبی که که در لیست تعیین‌کننده‌ی پایه‌ی کلاس ظاهر می‌شوند؛ سپس، اعضای داده غیرایستا، به ترتیبی که در تعریف کلاس اعلان شده‌اند.

### ۱.۴.۹ نمونه‌کد ناسازگار

در این نمونه‌کد، لیست مقداردهنده‌ی اولیه‌ی عضو متعلق به *C::C()* تلاش دارد ابتدا *someVal* را مقداردهی کند و سپس، *dependsOnSomeVal* را، با مقداری که به *someVal* وابسته است، مقداردهی اولیه کند. به‌دلیل آن‌که ترتیب اعلان متغیرهای عضو با ترتیب مقداردهنده‌های عضو همخوانی ندارد، تلاش برای خواندن مقدار *someVal*، مقدار تعیین‌نشده‌ای را که در *dependsOnSomeVal* ذخیره شده‌است، نتیجه می‌دهد.

```
class C
{
        int dependsOnSomeVal;
        int someVal;
        public:
                C(int val) : someVal(val), dependsOnSomeVal(someVal + 1) {}
};
```

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

### ۲.۴.۹ راه‌حل سازگار

این راه‌حل، ترتیب اعلان متغیرهای عضو کلاس را به‌گونه‌ای تغییر می‌دهد که وابستگی به‌خوبی بتواند در لیست مقداردهنده‌های اولیه‌ی عضو سازنده، مرتب شود.

```
class C
{
        int someVal;
        int dependsOnSomeVal;
        public:
                C(int val) : someVal(val), dependsOnSomeVal(someVal + 1) {}
};
```

منطقی است که مقداردهنده‌های اولیه به مقادیر مقداردهی‌های اولیه پیشین وابسته باشند.

### ۳.۴.۹ نمونه‌کد ناسازگار

در این نمونه‌کد، کلاس اشتقاقی *D* تلاش می‌کند کلاس پایه‌ی *B1* را، با مقداری که از کلاس پایه‌ی *B2* دریافت نموده‌است، مقداردهی اولیه کند. اگرچه، چون بنابر ترتیب اعلان در لیست تعیین‌کننده‌ی کلاس پایه، *B1* پیش از *B2* مقداردهی اولیه شده‌است، شاهد رفتاری تعریف‌نشده خواهیم بود.

```
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()) {}
};
```

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

### ۴,۴,۹ راه‌حل سازگار

این راه‌حل، به جای تکیه بر ترتیب مقداردهی اولیه‌ی کلاس‌های پایه، هردو کلاس را با استفاده از مقدار مشابهی، که از لیست پارامتر سازنده گرفته شده‌است، مقداردهی اولیه می‌کند.

```
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) {}
};
```

### ۵,۴,۹ ارزیابی خطر

| قانون | شدت/دقت | احتمال | هزینه‌ی اصلاح | اولویت | سطح |
|---|---|---|---|---|---|
| OOP53-CPP | متوسط | بعید | متوسط | P4 | L3 |

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکزماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

## ۵,۹ OOP54-CPP به‌خوبی به تخصیص خودکپی رسیدگی کنید

ممکن است تخصیص خودکپی[۶۹] در موقعیت‌های مختلف پیچیده‌ای اتفاق بیفتد. اما اساسا، تمامی این تخصیصات، مستلزم برخی تغییرات زیر است:

```
#include <utility>
struct S
{
        /* ... */
}
void f()
{
        S s;
        s = s; // Self-copy assignment
}
```

باید عملگرهای کپی کاربرمحور به‌درستی، پس از تخصیص خودکپی، بررسی شوند. به‌عنوان مثال، برای $x=y$ مقدار $y$ تغییر نمی‌کند. وقتی $&x = &y$ اعمال می‌شود، هیچ‌یک از دو متغیر $x$ و $y$ تغییر نمی‌یابند. یک پیاده‌سازی بومی تخصیص کپی می‌تواند منابع محلی شی را، در فرآیند نسخه‌برداری از پارامترهای داده‌شده، از بین ببرد. اگر پارامتر داده‌شده همان شی محلی باشد، از بین بردن منابع شی محلی منجر به غیرمعتبر کردن آنها خواهد شد. نسخه‌ی کپی‌شده از این منابع در وضعیت نامشخصی قرار می‌گیرد، که شرایط پس از اجرای کد[۷۰] را نقض می‌کند.

لازم است عملگر تخصیص کپی کاربرمحور، از تخصیص خودکپی از طریق رها کردن شی در یک وضعیت نامشخص، جلوگیری به‌عمل آورد. این مورد از طریق تست‌های تخصیص خودکپی، کپی و تعویض[۷۱]، یا سایر الگوهای طراحی مجازی و اصطلاحی، دست‌یافتنی است.

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

---

[۶۹] Self-copy assignment

[۷۰] Postcondition

[۷۱] Copy-and-swap

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

استاندارد C++ بر این نکته تاکید دارد که انواع، باید تضمین نمایند زمانی‌که تخصیص خودکپی به توابع کتابخانه‌ای الگوی استاندارد[۷۲] (STL) گذر می‌کند، در یک حالت ثابت شی را رها می‌نماید. به‌دلیل آن‌که اشیای نوع STL در مضامینی که به *CopyAssignable* نیاز است استفاده می‌شوند، انواع STL نیاز دارند تا تخصیص خودکپی را به‌خوبی رسیدگی کنند.

### ۱.۵.۹ نمونه‌کد ناسازگار

در این مثال، عملگر تخصیص کپی در برابر تخصیص خودکپی محافظت به‌عمل نمی‌آورد. اگر تخصیص خودکپی رخ دهد، *this->s1* حذف می‌شود. در نتیجه، *rhs.s1* نیز حذف خواهد شد. سپس، حافظه‌ی نامعتبر *rhs.s1*، به سازنده‌ی کپی *S* ارسال می‌شود، که می‌تواند ارجاع مجدد به یک اشاره‌گر نامعتبر را به‌دنبال داشته‌باشد.

```
#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;
                }
};
```

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

[۷۲] Standard Template Library (STL)

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

### ۲.۵.۹ راه‌حل سازگار (خودآزمایی)

این راه‌حل، با تست این‌که آیا پارامتر داده‌شده *this* است یا خیر، از برنامه در برابر تخصیص خودکپی، حفاظت به‌عمل می‌آورد. در صورتی‌که تخصیص خودکپی رخ دهد، آنگاه عملگر "-" هیچ کاری انجام نمی‌دهد؛ در غیر این‌صورت، کپی همانند مثال اصلی پردازش می‌شود.

```
#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;
		}
};
```

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکزماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

این راه‌حل، استثنائی قوی را برای تخصیص کپی تضمین نمی‌کند. مشخصا، اگر استثنائی هنگام ارزیابی عبارت *new* فراخوانده شود، بدان معنا است که *this* قبلا تغییر کرده‌است. اگرچه این راه‌حل، یک استثنای پایه را تضمین می‌کند، زیرا هیچ‌یک از منابع نشت نمی‌کنند و تمامی اعضای داده دارای مقادیر معتبر هستند.

### ۳.۵.۹ راه‌حل سازگار (کپی و تعویض)

این راه‌حل، با ایجاد یک شی موقت از *rhs*، که بعدا با **this* تعویض می‌شود، از تخصیص خودکپی جلوگیری به‌عمل می‌آورد. همچنین، راه‌حل مذکور، یک استثنای قوی را تضمین می‌کند. زیرا، اگر تخصیص منابع، در شرایطی‌که یک شی موقت در حال ایجاد است، منجر به وقوع یک استثنا شود، هرگز *swap()* فراخوانی نخواهد شد.

```
#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;
		}
};
```

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

### ۴,۵,۹ ارزیابی خطر

اگر به یک عملگر تخصیص کپی اجازه داده‌شود تا یک شی را خراب کند، می‌تواند باعث بروز یک رفتار تعریف‌نشده شود.

| قانون | شدت/دقت | احتمال | هزینه‌ی اصلاح | اولویت | سطح |
|---|---|---|---|---|---|
| OOP54-CPP | پایین | محتمل | بالا | P2 | L3 |

مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

## ۶.۹ OOP55-CPP برای دسترسی به اعضایی که وجود ندارند، از عملگرهای اشاره‌گر به عضو استفاده نکنید

عملگرهای اشاره‌گر به عضو[۷۳] "*." و "*->" جهت به‌دست آوردن یک شی یا تابع، که عضوی از یک شی زیربنایی است، استفاده می‌شوند. به‌عنوان مثال، در ادامه، دو روش کاربردی یکسان جهت فراخوانی تابع عضو *f()* بر شی *O* ارائه می‌شود.

```
struct S
{
        void f() {}
};
void func()
{
        S o;
        void (S::*pm)() = &S::f;
        o.f(); (o.*pm)();
}
```

فراخوانی فرم *o.f()*، از دسترسی عضو کلاس در زمان تفسیر، برای جستجوی آدرس تابع *S::f()* روی شی *o* استفاده می‌کند. فراخوانی فرم *(o.*pm)()*، از عملگر اشاره‌گر به عضو "*."، جهت فراخوانی تابع در آدرسی که توسط *pm* مشخص شده‌است، استفاد می‌کند. در هر دو مورد، شی *o*، شی صریح *this* داخل تابع عضو *S::f()* است.

اگر بخواهیم *pm-expression.*cast-expression* را به‌صورت *E1.*E2* به اختصار درآوریم، *E1*، عبارت شی نامیده می‌شود. اگر نوع پویای *E1* شامل عضوی نباشد که به *E2* ارجاع بدهد، رفتار، تعریف‌نشده است.

یک عبارت اشاره‌گر به عضو، به فرم *E1->*E2*، به فرم معادل آن تبدیل می‌شود *(*(E1)).*E2*. در نتیجه، استفاده از عبارت اشاره‌گر به عضو هرکدام از فرم‌ها، رفتار تعریف‌نشده‌ای را از خود بروز خواهد داد.

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

[۷۳] Pointer-to-member

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

به‌طور کلی، اگر عملوند دوم، اشاره‌گر تهی به مقدار عضو باشد، رفتار حاصل، یک رفتار تعریف‌نشده خواهد بود. زمانی‌که نوع پویای عملوند نخست شامل عضوی که عملوند دوم به آن ارجاع می‌دهد نیست (شامل استفاده از مقدار اشاره‌گر به عضو تهی به عنوان عملوند دوم) از عبارت اشاره‌گر به عضو استفاده نکنید.

### ۲,۶,۹ نمونه‌کد ناسازگار

در این مثال، یک شی اشاره‌گر به عضو، از *D::g* گرفته می‌شود. اما سپس، به *B::** تبدیل رو به بالا (upcast)[۷۴] (تبدیل از نوع اشتقاقی بیشتر به نوع اشتقاقی کمتر) می‌شود. زمانی‌که یک شی، که نوع پویای آن *D* است، فراخوانی می‌شود، فراخوانی اشاره‌گر به عضو به‌خوبی تعریف می‌شود. اگرچه، نوع پویای شی زیربنایی، *B* است، که رفتار تعریف‌نشده‌ای را نتیجه می‌دهد.

```
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;
}
```

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

[۷۴] Upcast

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

### ۲,۶,۹ راه‌حل سازگار

در این راه‌حل، تبدیل رو به بالا حذف می‌شود، که کد ابتدایی را به‌صورت بدفرم تفسیر می‌کند و بر حذف مشکل زیربنایی *B::g()* تاکید دارد. راه‌حل سازگار فرض می‌کند که هدف برنامه‌نویس، استفاده‌ی درست از نوع پویا برای شی زیربنایی است.

```
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;
}
```

### ۳,۶,۹ نمونه‌کد ناسازگار

در این مثال، یک مقدار اشاره‌گر به عضو تهی، به‌عنوان عملوند دوم، به یک عبارت اشاره‌گر به عضو، گذر می‌کند، که یک رفتار تعریف‌نشده را نتیجه می‌دهد.

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکزماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

```
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;
}
```

### ۴,۶,۹ راه‌حل سازگار

در این راه‌حل، *gptr*، به‌درستی، به جای مقدار پیش‌فرض *nullptr*، با یک مقدار اشاره‌گر به عضو، مقداردهی اولیه می‌شود.

```
struct B
{
	virtual ~B() = default;
};
struct D : B
{
	virtual ~D() = default;
	virtual void g()
	{
		/* ... */
	}
};
```

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

```
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;
}
```

### ۵,۶,۹ ارزیابی خطر

| قانون | شدت/دقت | احتمال | هزینه‌ی اصلاح | اولویت | سطح |
|---|---|---|---|---|---|
| OOP55-CPP | بالا | محتمل | بالا | P6 | L2 |

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

## ۷,۹ OOP56-CPP به نیازمندی‌های نگهدارنده‌ی جایگزینی احترام بگذارید

توابع نگهدارنده[۷۵] *terminate_handeler*، *new_handeler* و *unexpected_handeler* می‌توانند به‌صورت عمومی، با پیاده‌سازی‌های دستی جایگزین شوند. به‌عنوان مثال، یک برنامه می‌تواند با فراخوانی *std::set_terminate()*، یک نگهدارنده‌ی خاتمه‌ی دستی تنظیم کند، ممکن است نگهدارنده‌ی خاتمه‌ی دستی، اختتام را برای رسیدگی‌های بعدی ثبت کند.

در موارد خاص (توابع جایگزینی، توابع نگهدارنده، عملیات‌های اجرایی بر نوعی که برای معرفی اجزای الگوی کتابخانه‌ی استاندارد استفاده می‌شوند)، کتابخانه‌ی استاندارد C++ به مؤلفه‌هایی، که توسط برنامه‌ی C++ عرضه می‌شوند، بستگی دارد.

- یک *new_handler* باید یکی از موارد زیر را انجام دهد:
  - حافظه‌ی بیشتری برای تخصیص در دسترس قرار دهد و سپس بازگرداند.
  - یک استثنا نوع *bad_alloc* و یا یک کلاس اشتقاقی از *bad_alloc* را ایجاد کند.
  - اجرای برنامه را، بدون بازگشتن به فراخواننده، متوقف کند.
- یک *terminate_handler* باید اجرای برنامه را، بدون بازگشتن به فراخواننده، متوقف کند.
- یک *unexpected_handler* نباید بازگردد.

### ۱,۷,۹ نمونه‌کد ناسازگار

در این مثال، یک *new_handler* سعی دارد وقتی حافظه‌ی مدیر حافظه‌ی پویا به اتمام می‌رسد، منابع قابل نجات را آزاد کند. اگرچه این مثال، شرایطی را که در آن تمامی منابع قابل نجات، نجات یافته‌اند و هنوز حافظه‌ی کافی برای ارضای درخواست‌های تخصیص وجود ندارد، در نظر نگرفته‌است. نگهدارنده‌ی جایگزین، به جای خاتمه‌ی نگهدارنده‌ی جایگزین با یک استثنای نوع *std::bad_alloc*، یا اختتام برنامه بدون بازگشتن به فراخواننده، به‌صورت طبیعی بازمی‌گردد. در شرایط کمبود حافظه، یک حلقه‌ی بی‌نهایت با اجرای پیش‌فرض

[۷۵] Handeler

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

*operator new()*:: روی خواهد داد. به‌دلیل آن‌که چنین موقعیتی در عمل بسیا نادر است، احتمال زیادی وجود دارد که این اشکال[۷۶] در سناریوهای تست معمولی، کشف نشود.

```
#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);
	// ...
}
```

### ۲.۷.۹ راه‌حل سازگار

در این راه‌حل، *custom_new_handler()* از مقدار بازگشتی *reclaim_resources()* استفاده می‌کند. اگر مقدار صفر را بازگرداند، در نتیجه، میزان حافظه برای موفقیت‌آمیز بودن *operator new*، ناکافی است. از این‌رو، یک استثنا نوع *std::bad_alloc* ایجاد می‌شود تا نیازهای نگهدارنده‌ی جایگزین را مرتفع سازد.

```
#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);
	// ...
}
```

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

[۷۶] Bug

### ۳.۷.۹ ارزیابی خطر

عدم موفقیت در برطرف‌سازی نیازهای نگهدارنده‌ی جایگزین، باعث ایجاد رفتار تعریف‌نشده می‌شود.

| قانون | شدت/دقت | احتمال | هزینه‌ی اصلاح | اولویت | سطح |
|---|---|---|---|---|---|
| OOP56-CPP | پایین | محتمل | بالا | P2 | L3 |

مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

## ۸,۹ OOP57-CPP توابع عضو خاص و عملگرهای دچار سربارشده را به توابع کتابخانه‌ی استاندارد C ترجیح دهید

چندین تابع کتابخانه‌ی استاندارد C، عملیات‌های بایتی روی اشیا انجام می‌دهند. به‌عنوان مثال، *std::mememp()*، بایت‌های تشکیل‌دهنده‌ی دو شی را با یکدیگر مقایسه می‌کند؛ و *std::memcpy()*، بایت‌های تشکیل‌دهنده‌ی یک شی را در بافر مقصد کپی می‌کند. اگرچه، این عملیات در بعضی انواع اشیا، منجر به بروز رفتار تعریف‌نشده و غیرطبیعی در برنامه می‌شود.

یک کلاس قابل کپی بدیهی کلاسی است که:

- هیچ سازنده‌ی کپی غیربدیهی نداشته‌باشد.
- هیچ سازنده‌ی انتقال غیربدیهی نداشته‌باشد.
- هیچ عملگر انتساب کپی غیربدیهی نداشته‌باشد.
- هیچ عملگر انتقال و انتساب غیربدیهی نداشته‌باشد.
- و دارای یک مخرب بدیهی باشد.

از *std::memset()* برای مقداردهی اولیه‌ی کلاس غیربدیهی استفاده نکنید، زیرا ممکن است به‌درستی، مقدار ارائه‌شده‌ی شی را مقداردهی اولیه نکند. از *std::memcpy()* (یا سایر توابع کپی بایتی) برای مقداردهی اولیه‌ی کپی یک کلاس غیربدیهی استفاده نکنید، زیرا ممکن است به‌درستی، مقدار ارائه‌شده برای کپی شی را مقداردهی اولیه نکند. از *std::memcmp()* (یا سایر توابع مقایسه‌ی بایتی) برای مقایسه‌ی دو شی از کلاس لایه‌ای غیر استاندارد استفاده نکنید، زیرا ممکن است به‌درستی، مقدار ارائه‌شده‌ی اشیا را مقایسه نکند. در تمام موارد بهتر است از جایگزین‌ها استفاده شود.

### ۱,۸,۹ نمونه‌کد ناسازگار

در این مثال، یک شی از کلاس غیربدیهی از طریق فراخوانی سازنده‌ی پیش‌فرض خود، مقداردهی اولیه شده‌است. اما بعدا، با استفاده از *std::memset()*، مجددا به حالت پیش‌فرض خود، مقداردهی اولیه می‌شود. طی

مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکزماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

این فرآیند، شی به‌درستی مقداردهی اولیه‌ی مجدد نمی‌شود. مقداردهی آغازین مجدد نامناسب باعث می‌شود که ویژگی‌های کلاس برای استفاده‌های بعدی شی، باقی نمانند.

```
#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;
}
```

نمونه‌کد ناسازگار بالا با EXP62-CPP سازگار است. به بیت‌های ارائه‌ی یک شی، که بخشی از ارائه‌ی مقدار شی نیستند، دسترسی پیدا نکنید. زیرا، تمام بیت‌هایی که در بخش ارائه‌ی مقدار هستند، در ارائه‌ی شی C نیز استفاده می‌شوند.

### ۲.۸.۹ راه‌حل سازگار

در این راه‌حل، عملیات کپی و تعویض مقداردهی اولیه‌شده‌ی پیش‌فرضی به نام *clear ()*، جایگزین فراخوانی *std::memset()* شده‌است. این عملیات تضمین می‌کند که شی به‌درستی، به حالت پیش‌فرض خود مقداردهی اولیه می‌شود، و برای انواعی که دارای عملگرهای تخصیص بهینه هستند، که از پاک کردن تمامی اعضای داده‌ی در حال تخصیص شی بازمی‌ماند، به‌درستی رفتار می‌کند.

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه‌ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

```
#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;
}
```

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه‌ای

### ۳.۸.۹ نمونه‌کد ناسازگار

در این مثال، از *std::memcpy()* برای ایجاد یک کپی از شی بدیهی نوع C استفاده می‌شود. اگرچه، به‌دلیل آن‌که هر نمونه از شی تلاش دارد تا *int* * را در *C::~C()* حذف کند، ممکن است آسیب‌پذیری‌های دوچندان رخ دهد. زیرا، همان مقدار اشاره‌گر، در *c2* کپی خواهد شد.

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکزماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

```
#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));
}
```

### ۴,۸,۹ راه‌حل سازگار

در این راه‌حل، C یک عملگر تخصیص تعریف می‌کند، که به جای فراخوانی *std::memcpy()* استفاده می‌شود.

```
class C
{
	int *i;
	public:
		C(): i(nullptr) {}
		~C()
		{
			delete i;
		}
		void set(int val)
		{
			if (i)
			{
				delete i;
			}
			i = new int{val};
		}
```

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

```
        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;
}
```

### ۵,۸,۹ نمونه‌کد ناسازگار

در این مثال، از *std::memcmp()* برای مقایسه‌ی دو شی لایه‌ای غیراستاندارد استفاده می‌شود. از آنجاکه *std::memcmp ()*، یک مقایسه‌ی بایتی از ارائه‌ی اشیا انجام می‌دهد، اگر پیاده‌سازی از یک اشاره‌گر جدول مجازی به‌عنوان بخشی از ارائه‌ی شی استفاده کند، این روش، اشاره‌گرهای جدول مجازی را باهم مقایسه خواهد نمود. اگر نوع پویای c1 یا c2، یک کلاس اشتقاقی از C باشد، مقایسه بدون توجه به ارائه‌ی مقدار هر یک از اشیا، شکست خواهد خورد.

به‌دلیل آن‌که یک تابع مجازی، بخشی از ارائه‌ی مقدار شی محسوب نمی‌شود، مقایسه‌ی آن با *std::memcmp()*، EXP62-CPP را نیز نقض می‌کند. به بیت‌های ارائه‌ی یک شی، که بخشی از ارائه‌ی مقدار شی نیستند، دسترسی پیدا نکنید.

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

```
#include <cstring>
class C
{
	int i;
	public:
		virtual void f();
		// ...
};
void f(C &c1, C &c2)
{
	if (!std::memcmp(&c1, &c2, sizeof(C)))
	{
		// ...
	}
}
```

### ۶,۸,۹ راه‌حل سازگار

در این راه‌حل، C یک عملگر برابری را تعریف می‌کند که به جای *std::memcmp()* استفاده می‌شود. این راه‌حل تضمین می‌کند که ارائه‌ی مقدار شی، تنها هنگام مقایسه در نظر گرفته می‌شود.

```
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)
	{
		// ...
	}
}
```

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

### ۷،۸،۹ ارزیابی خطر

نتیجه‌ی اکثر نقض‌های این قانون، رفتار غیرطبیعی در برنامه خواهد بود. اگرچه، ممکن است نوشتن مجدد جزئیات پیاده‌سازی ارائه‌ی شی، منجر به آسیب‌پذیری‌های اجرای کد شود.

| قانون | شدت/دقت | احتمال | هزینه‌ی اصلاح | اولویت | سطح |
|---|---|---|---|---|---|
| OOP57-CPP | بالا | محتمل | بالا | P6 | L2 |

مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

## ۹,۹ OOP58-CPP عملیات‌های کپی نباید شی منبع را جهش دهند

انتظار می‌رود عملیات‌های کپی (سازنده‌های کپی و عملگرهای تخصیص کپی)، خواص مهم شی منبع را در شی مقصد کپی کنند. در نتیجه، شی مقصد یک کپی از نسخه‌ی منبع خواهد بود. این‌که چه چیزی در زمره‌ی خواص مهم یک شی در نظر گرفته‌شود، به نوع آن شی بستگی دارد، اما برای انواعی که عملگرهای مقایسه یا برابری را عرضه می‌کنند، شامل هر ویژگی مورداستفاده برای عملیات‌های مقایسه می‌شود. این انتظارات، مفروضاتی را در کد ایجاد می‌کند که طی آنها یک عملیات کپی، یک شی مقصد حاوی ارائه‌ی مقداری برابر با ارائه‌ی مقدار شی منبع را نتیجه خواهد داد. نقض این فرض باعث بروز رفتار تعریف‌نشده می‌شود.

در حالت ایده‌ال، عملگر کپی باید دارای یک امضای اصطلاحی باشد. این امضا برای سازنده‌های کپی و عملگرهای تخصیص کپی، به‌ترتیب عبارت است از *T(const T&);* و *T& operator=(const T&);*. عملگرهای سازنده‌ی تخصیص کپی، که از امضای اصطلاحی استفاده نمی‌کنند، به‌ترتیب، نیازمندی‌های مفهوم *CopyConstructible* یا *CopyAssignable* را برآورده نمی‌سازند. این امر مانع از استفاده از نوع با قابلیت کتابخانه‌ی استاندارد معمولی می‌شود.

هنگام پیاده‌سازی یک عملگر کپی، هیچ یک از اعضای قابل مشاهده‌ی خارجی عملوند شی منبع یا اطلاعات قابل دسترسی عمومی را جهش یا تغییر ندهید.

### ۱,۹,۹ نمونه‌کد ناسازگار

در این مثال، عملیات‌های کپی برای *A*، عملوند منبع را از طریق بازنشانی متغیر عضو خود (*m*) به صفر، جهش می‌دهند. وقتی *std::fill()* فراخوانده می‌شود، اولین مؤلفه‌ی کپی‌شده، مقدار اصلی *obj.m* (۲) را خواهد داشت. این در حالی است که مقدار *obj.m*، به صفر تنظیم می‌شود. تمامی نُه کپی، مقدار صفر را باز خواهند گرداند.

```
#include <algorithm>
#include <vector>
class A
{
        mutable int m;
        public:
                A() : m(0) {}
                explicit A(int m) : m(m) {}
```

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکزماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

```
        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);
}
```

### ۲.۹.۹ راه‌حل سازگار

در این راه‌حل، عملگر کپی برای *A*، دیگر عملوند منبع را جهش نمی‌دهد، بلکه تضمین می‌کند که بردار، شامل کپی‌های یکسانی از *obj* است. در عوض، عملگرهای انتقال *A*، داده شده‌است، تا زمانی‌که انجام جهش، امن است، آن را انجام دهند.

```
#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;
                }
```

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکزماهر
مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

```
        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);
}
```

### ۳.۹.۹ ارزیابی خطر

عملیات‌های کپی، که عملوند منبع یا حالت عمومی را جهش می‌دهند، می‌توانند به رفتار غیرمنتظره‌ی برنامه منجر شوند. استفاده از چنین نوعی در نگهدارنده‌ی STL یا در الگوریتم، می‌تواند باعث بروز رفتار تعریف‌نشده شود.

| قانون | شدت/دقت | احتمال | هزینه‌ی اصلاح | اولویت | سطح |
|---|---|---|---|---|---|
| **OOP58-CPP** | پایین | احتمال زیاد | پایین | P9 | L2 |

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران

مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

# فصل دهم

# همزمانی (CON)

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکزماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

## ۱,۱۰ CON50-CPP یک میوتکس قفل‌شده را از بین نبرید

اشیای میوتکس جهت محافظت از داده‌های به اشتراک گذاشته‌شده، در برابر دسترسی همزمان به آنها استفاده می‌شوند. اگر، در حالی‌که یک نخ بلاک‌شده در انتظار قفل به‌سر می‌برد، یک شی میوتکس از بین برود، دیگر، ناحیه‌های بحرانی و داده‌های به اشتراک گذاشته‌شده، تحت پوشش نخواهند بود.

اگر برنامه‌ای یک شی میوتکس را، که تحت تملک این یک نخ است، از بین ببرد و یا نخی که صاحب شی میوتکس است از بین برود، رفتار برنامه، تعریف‌نشده خواهد بود. چنین عبارتی برای *std::recursive_mutex* و *std::shared_timed_mutex* نیز صادق است. در واقع، از بین بردن یک شی میوتکس، که یک نخ در انتظار آن به‌سر می‌برد، باعث بروز رفتار تعریف‌نشده‌ی برنامه خواهد شد.

### ۱,۱,۱۰ نمونه‌کد ناسازگار

این مثال، چندین نخ ایجاد می‌کند که هرکدام، تابع *do_work()* را فرا می‌خوانند. تابع مذکور، عدد منحصر به‌فردی را به‌عنوان ID از خود عبور می‌دهد. متاسفانه این کد دارای شرایط رقابتی است و امکان تخریب میوتکس را، حتی وقتی که هنوز در تملک است، فراهم می‌کند. زیرا ممکن است پیش از آن‌که تمامی نخ‌ها خارج شوند، *start_threads()*، مخرب میوتکس را فراخوانی کند.

```
#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);
	}
}
```

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

### ۲,۱,۱۰ راه‌حل سازگار

این راه‌حل، حالت رقابتی را با گسترش عمر میوتکس از بین می‌برد.

```
#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);
	}
}
```

### ۳,۱,۱۰ راه‌حل سازگار

این راه‌حل، پیش از آن‌که مخرب میوتکس فرا خوانده شود، نخ‌ها را به یکدیگر متصل می‌سازد و حالت رقابتی را از بین می‌برد.

```
#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);
	}
```

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

```
        for (size_t i = 0; i < maxThreads; ++i)
        {
                threads[i].join();
        }
}
```

### ۴.۱.۱۰ ارزیابی خطر

ممکن است تخریب میوتکسی که در حال قفل‌شدن است، جریان کنترلی نامعتبر و خرابی داده را با خود به‌همراه داشته‌باشد.

| قانون | شدت/دقت | احتمال | هزینه‌ی اصلاح | اولویت | سطح |
|---|---|---|---|---|---|
| CON50-CPP | متوسط | محتمل | بالا | P4 | L3 |

مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

## ۲,۱۰ CON51-CPP مطمئن شوید که قفل‌های برگزارشده‌ی فعال، در شرایط استثنایی آزاد می‌شوند

ممکن است میوتکس‌هایی که برای محافظت در برابر دسترسی به داده‌های به اشتراک گذاشته‌شده استفاده می‌شوند، توسط تابع عضو *lock()*، قفل و با تابع عضو *unlock()*، آزاد شوند. اگر استثنایی میان فراخوانی *lock()* و *unlock()* رخ دهد، آن استثنا جریان کنترل را به‌گونه‌ای تغییر می‌دهد که *unlock()* فرا خوانده نشود و میوتکس، در حالت قفل باقی بماند. در نتیجه، هیچ بخش بحرانی که توسط آن میوتکس حفاظت می‌شود، اجازه‌ی اجرا نخواهد داشت. این امر به احتمال فراوان، منجر به وقوع بن‌بست خواهد شد.

وقوع یک استثنا نباید به یک میوتکس اجازه دهد تا به‌طور نامحدود، در حالت قفل باقی بماند. اگر میوتکسی قفل باشد و استثنائی در بخش بحرانی تحت حفاظت آن میوتکس اتفاق بیفتد، باید میوتکس به‌عنوان بخشی از رسیدگی به استثنا، پیش از وقوع مجدد و یا ادامه‌ی اجرا، از حالت قفل خارج شود. مگر آن‌که جریان کنترل، متعاقبا میوتکس را از حالت قفل خارج کند.

زبان C++، کلاس‌های قفل *lock_guard*، *unique_lock* و *shared_lock* را فراهم آورده‌است، که می‌توانند توسط یک میوتکس، مقداردهی اولیه شوند. در سازنده‌ی آن، شیء قفل، میوتکس را قفل می‌کند، و در مخرب آن، میوتکس را آزاد می‌نماید. کلاس *lock_guard*، یک احاطه‌کننده‌ی RAII پیرامون میوتکس ایجاد می‌کند. *shared_lock* و *unique_lock* نیز از RAII در کاربردهای دیگر، مانند کنترل دستی استراتژی قفل، استفاده می‌کنند. کلاس *unique_lock*، از کپی شدن قفل ممانعت به‌عمل می‌آورد. اگرچه، اجازه می‌دهد مالکیت قفل به قفل دیگری اعطا شود. کلاس *shared_lock* نیز اجازه می‌دهد که میوتکس بین چند قفل به اشتراک گذاشته‌شود. اگر استثنائی برای هر سه کلاس به‌وقوع بپیوندد و کنترل جریان را خارج از حوزه‌ی قفل به‌دست گیرد، مخرب، میوتکس را آزاد می‌کند و برنامه می‌تواند به‌صورت عادی کار خود را ادامه دهد.

### ۱,۲,۱۰ نمونه‌کد ناسازگار

این کد، داده‌های به اشتراک گذاشته‌شده تولید و نواحی بحرانی آنها را با میوتکس، قفل می‌کند. زمانی‌که به پایان می‌رسد، میوتکس‌ها را آزاد می‌سازد. با این‌وجود، اگر در حین تولید داده‌های به اشتراک گذاشته‌شده استثنایی رخ دهد، میوتکس به حالت قفل باقی می‌ماند.

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

```
#include <mutex>
void manipulate_shared_data(std::mutex &pm)
{
	pm.lock();
	// Perform work on shared data.
	pm.unlock();
}
```

### ۲.۲.۱۰ راه‌حل سازگار (آزادسازی دستی)

این راه‌حل، هر استثنایی را، که بر داده‌های به اشتراک گذاشته‌شده به‌وقوع بپیوندد، گرفته و پیش از وقوع دوباره‌ی استثنا، آن را آزاد می‌سازد.

```
#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
}
```

### ۳.۲.۱۰ راه‌حل سازگار (شی قفل)

این راه‌حل، از یک شی *lock_guard* استفاده می‌کند تا تضمین نماید که حتی در صورت وقوع یک استثنا و بدون اتکا به دستگاه رسیدگی به استثنا و کنترل دستی منابع، میوتکس آزاد باقی می‌ماند.

```
#include <mutex>
void manipulate_shared_data(std::mutex &pm)
{
	std::lock_guard<std::mutex> lk(pm);
	// Perform work on shared data.
}
```

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

### ۴,۲,۱۰ارزیابی خطر

اگر در زمانی‌که میوتکسی قفل است استثنا رخ دهد، بن‌بست به‌وقوع خواهد پیوست.

| قانون | شدت/دقت | احتمال | هزینه‌ی اصلاح | اولویت | سطح |
|---|---|---|---|---|---|
| CON51-CPP | پایین | محتمل | پایین | P6 | L2 |

مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

## ۳,۱۰ CON52-CPP به هنگام دسترسی به فیلدهای بیتی چندین نخ، از رقابت‌های داده‌ای بپرهیزید

ممکن است یک نخ هنگام دسترسی به فیلد بیتی، سهوا به بیت فیلدی حافظه‌ی مجاور دسترسی پیدا کند. زیرا نیاز است مفسرها تا زمانی‌که دارای ظرفیت لازم هستند، جهت ذخیره‌ی چندین فیلد بیتی مجاور در یک مخزن به کار گرفته شوند. در نتیجه، ممکن است رقابت داده‌ای، نه تنها در فیلد بیتی که توسط چندین نخ قابل دست‌یابی است، بلکه در سایر فیلدهای بیتی که همان بایت یا کلمه را به اشتراک می‌گذراند، وجود داشته‌باشد. تشخیص مسأله قدری دشوار است، زیرا ممکن است تشخیص آن‌که همان مکان از حافظه توسط چندین نخ در حال تغییر است، واضح نباشد.

یکی از راه‌های جلوگیری از رقابت داده‌ای در برنامه‌های همزمان، استفاده از میوتکس است. وقتی یک میوتکس توسط تمامی نخ‌ها مشاهده می‌شود، می‌تواند یک دسترسی امن به یک شی به اشتراک گذاشته‌شده داشته‌باشد. اگرچه، وقتی میوتکس توسط نخ دسترسی کنترل نشود، هیچ تضمینی در ارتباط با سایر اشیایی که ممکن است در دسترس قرار گیرند، ارائه نمی‌دهند. متاسفانه، هیچ روش قابل حملی برای مشخص کردن این‌که کدام فیلد بیتی ممکن است همراه با فیلد بیتی موردنظر و مطلوب ذخیره شده‌باشد، وجود ندارد.

یک روش دیگر، قرار دادن یک عضو غیر فیلد بیتی بین هر دو فیلد بیتی است تا اطمینان یابیم که هر فیلد بیتی، تنها فیلدی است که داخل واحد ذخیره‌سازی خود مورد دسترسی قرار گرفته‌است. این تکنیک به‌طور موثر تضمین می‌کند که هیچ دو فیلد بیتی به‌صورت همزمان در دسترس قرارنمی‌گیرند.

### ۱,۳,۱۰ نمونه‌کد ناسازگار

این امکان وجود دارد که فیلدهای بیتی فیلد مجاور دریک مکان از حافظه ذخیره شوند. در نتیجه، همان‌گونه که در این کد مشاهده می‌شود، تغییر فیلدهای بیتی مجاور در چندین نخ متفاوت، حاکی از یک رفتار تعریف‌نشده است.

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

```
struct MultiThreadedFlags
{
        unsigned int flag1 : 2;
        unsigned int flag2 : 2;
};
MultiThreadedFlags flags;
void thread1()
{
        flags.flag1 = 1;
}
void thread2()
{
        flags.flag2 = 2;
}
```

به‌عنوان مثال، مراتب دستوری زیر امکان‌پذیر است.

```
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
```

### ۲,۳,۱۰ راه‌حل سازگار (فیلد بیتی C++ 11 و سپس، میوتکس)

این راه‌حل، تمام دسترسی‌ها به پرچم را توسط یک میوتکس پوشش می‌دهد و متعاقباً از رقابت داده‌ای ممانعت به‌عمل می‌آورد.

```
#include <mutex>
struct MultiThreadedFlags
{
        unsigned int flag1 : 2;
        unsigned int flag2 : 2;
};
struct MtfMutex
{
        MultiThreadedFlags s;
        std::mutex mutex;
};
MtfMutex flags;
```

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

```
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;
}
```

### ۳،۳،۱۰راه‌حل سازگار( C++ 11 )

در این راه‌حل، دو نخ به‌طور همزمان، عضو غیر فیلد بیتی متمایز از یک ساختار را تغییر می‌دهند. به‌دلیل آن‌که اعضای دو بایت متفاوت را در حافظه اشغال می‌کنند، به هیچ حفاظتی در برابر همزمانی نیاز نیست.

```
struct MultiThreadedFlags
{
        unsigned char flag1;
        unsigned char flag2;
};
MultiThreadedFlags flags;
void thread1()
{
        flags.flag1 = 1;
}
void thread2()
{
        flags.flag2 = 2;
}
```

با قطعیت نسبی می‌توان گفت که *flag1* و *flag2* در یک کلمه ذخیره شده‌اند. اگر هر دو تخصیص در جای‌گذاری بیتی زمان‌بندی نخ، که با هر دو ذخایری که یکی پس از دیگری اتفاق می‌افتند و خاتمه می‌یابند، رخ دهد، شاید استفاده از مفسری که با نسخه‌های پیشین استاندارد مطابقت دارد، این امکان را به‌وجود آورد که تنها یکی از پرچم‌ها به‌صورت مطلوب مقدار گیرد و پرچم دیگر، مقدار قبلی خود را حفظ کند. زیرا، هر دو عضو توسط یک کلمه (کوچکترین مؤلفه‌ای که پردازنده می‌تواند با آن کار کند) ارائه شده‌اند.

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

### ۴,۳,۱۰ ارزیابی خطر

با وجود این‌که پنجره‌ی رقابت باریک است، یک تخصیص یا اصطلاح می‌تواند بنابر داده‌های به اشتباه تفسیرشده، به‌نادرستی ارزیابی شود. در نتیجه، منجر به حالت اجرایی خراب و یا افشای ناخواسته‌ی اطلاعات گردد.

| قانون | شدت/دقت | احتمال | هزینه‌ی اصلاح | اولویت | سطح |
|---|---|---|---|---|---|
| CON52-CPP | متوسط | محتمل | متوسط | P8 | L2 |

مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکزماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

## ۴,۱۰ CON53-CPP با قفل کردن به ترتیب از پیش تعیین‌شده، از بن بست اجتناب کنید

از میوتکس‌ها جهت جلوگیری از رقابت داده‌ای چند نخ، که به‌صورت همزمان به یک منبع به اشتراک گذاشته‌شده دسترسی پیدا می‌کنند، استفاده می‌شود. گاهی، چندین نخ قفل یکدیگر را هنگام قفل کردن میوتکس‌ها نگاه می‌دارند. در نتیجه، برنامه دچار بن‌بست می‌شود. چهار شرط لازم جهت وقوع بن‌بست عبارتند از:

- محرومیت مشترک (حداقل باید یک منبع غیر قابل اشتراک نگهداری شود)
- نگهداری و انتظار (یک نخ باید منبعی را که منتظر آزادسازی سایر منابع است، نگه دارد)
- عدم پیش‌دستی (نمی‌توان منابعی را، که نخی در حال استفاده از آنها است، از نخ گرفت)
- انتظار چرخه‌ای (یک نخ باید منتظر منبعی که در اختیار نخ دیگر است، که آن هم در انتظار منبع تحت اختیار نخ اول است، بماند).

بن‌بست‌ها به تمامی چهار شرط فوق جهت وقوع نیاز دارند. بنابراین، جلوگیری از بن‌بست‌ها نیازمند جلوگیری از یکی از شروط مذکور است. یک راه‌حل ساده، قفل کردن میوتکس‌ها در ترتیبی از پیش تعیین‌شده، جهت جلوگیری از انتظار چرخه‌ای است.

### ۱,۴,۱۰ نمونه کد ناسازگار

رفتار این کد، به محیط در زمان اجرا و نیز زمانبندی پلتفرم بستگی دارد. اگر نخ *thr1* سعی کند میوتکس *ba2* را در زمانی که که نخ *thr2* تلاش دارد میوتکس *ba1* را در تابع *deposit()* قفل کند، قفل نماید، برنامه مشکوک به وقوع بن‌بست خواهد بود.

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

```
#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();
}
```

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

### ۲,۴,۱۰ راه‌حل سازگار (مرتب‌سازی دستی)

این راه‌حل، با ایجاد یک ترتیب از پیش تعیین‌شده برای قفل کردن در تابع *deposit()*، شرط انتظار چرخه‌ای را حذف می‌کند. هر نخ بر اساس *BankAccount ID*، که در زمان مقداردهی اولیه‌ی شی *BankAccount* تنظیم شده‌است، قفل خواهد کرد.

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

```
#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.
```

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکزماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

```
        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();
}
```

### ۳.۴.۱۰ راه‌حل سازگار (std::lock())

این راه‌حل، از امکانات STL استفاده می‌کند تا تضمین نماید که بن‌بست به‌دلیل انتظار چرخه‌ای روی نمی‌دهد. تابع *std::lock()*، یک عدد متغیر از اشیای قابل قفل را دریافت و سعی می‌کند آنها را به‌گونه‌ای قفل کند که بن‌بست رخ ندهد. در پیاده‌سازی‌های، معمول این عمل با استفاده از ترکیب *lock()*، *try_lock()*، و *unlock()* برای قفل کردن شی و بازگشت در صورت عدم نیاز به قفل، انجام می‌گیرد که ممکن است عملکرد بدتری نسبت به راه‌حل مبتنی بر ترتیب از پیش تعیین‌شده داشته‌باشد.

```
#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;
                }
};
```

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

```
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();
}
```

### ۴.۴.۱۰ ارزیابی خطر

بن‌بست، از پیشروی نخ‌ها، که منجر به توقف برنامه می‌شوند، جلوگیری می‌کند. اگر مهاجم بتواند شرایط بن‌بست ایجاد نماید، یک حمله‌ی انکار خدمات به وقوع خواهد پیوست.

| قانون | شدت/دقت | احتمال | هزینه‌ی اصلاح | اولویت | سطح |
|---|---|---|---|---|---|
| CON53-CPP | پایین | محتمل | متوسط | P4 | L3 |

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکزماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

## ۵.۱۰ CON54-CPP توابعی را که ممکن است به‌صورت جعلی در یک حلقه پدیدار شوند، بپوشانید

توابع عضو *wait_for ()*، *wait ()*، و *wait_until* از کلاس *std::condition_variable*، موقتا از در اختیار داشتن میوتکس صرف‌نظر می‌کنند تا سایر نخ‌هایی که ممکن است درخواست میوتکس داشته‌باشند، بتوانند اقدام نمایند. این توابع باید همیشه توسط کدی که توسط میوتکس حفاظت می‌شود، فراخوانی گردند. نخ در حال انتظار، تنها پس از آن‌که مطلع می‌شود (در واقع، زمانی‌که توابع عضو *notify_one()* یا *notify_all()* توسط سایر نخ‌ها فراخوانی می‌شوند)، اجرا را ادامه می‌دهد.

تابع *wait()* باید توسط حلقه‌ای که برقراری گزاره‌ی شرطی را بررسی می‌کند، فراخوانی شود. گزاره‌ی شرط اصطلاحی است که بر اساس آن، اگر نخی بخواهد اجرا را ادامه دهد، متغیرهای یک تابع باید برای آن نخ درست باشند. نخ با *wait_for ()*، *wait ()*، *wait_until*، یا مکانیزم دیگری متوقف می‌شود و بعدا، احتمالا زمانی‌که گزاره‌ی شرط درست باشد و نخ نیز مطلع شود، ادامه می‌یابد.

```
#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.
}
```

مکانیزم اطلاع، نخ در حال انتظار را مطلع می‌کند و به آن اجازه می‌دهد تا گزاره‌ی شرط خود را چک نماید. فراخوانی *notify_all()* در سایر نخ‌ها نمی‌تواند دقیقا مشخص کند کدام نخ ادامه پیدا خواهد کرد. اصطلاحات گزاره‌ی شرط به نخ‌های مطلع اجازه می‌دهد که مشخص نمایند آیا لازم است پس از دریافت اطلاع‌رسانی، فرآیند را از سر گیرد یا خیر.

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

### ۱.۵.۱۰ نمونه کد ناسازگار

این کد، یک لیست پیوندی را نمایش داده و در صورت خالی نبودن لیست، نخی را به مؤلفه‌های آن لیست، اختصاص می‌دهد. این نخ با استفاده از *wait()*، اجرا را متوقف می‌کند و وقتی مطلع گردد که احتمالا لیست دارای اجزایی است که باید مصرف شوند، کار خود را از سر خواهد گرفت. ممکن است حتی در صورت خالی بودن لیست، نخ مطلع گردد. زیرا، شاید نخ مطلع‌کننده، از *notify_all()*، که تمامی نخ‌ها را مطلع می‌سازد، استفاده کرده‌باشد. غالبا، آگاه‌سازی توسط *notify_all()* به *notify_one()* ترجیح داده‌می‌شود. به هنگام استفاده از متغیرهای شرطی، امنیت و زنده بودن نخ را حفظ نمایید.

معمولا، یک گزاره‌ی شرط یک نفی از اصطلاح شرطی موجود در حلقه است. در این مثال، (*list->next != nullptr*)، یک گزاره‌ی شرطی جهت حذف یک عنصر از لیست پیوندی است. این در حالی است که اصطلاح شرطی برای شرط حلقه، (*list->next == nullptr*) است.

این مثال، فراخوانی *wait()* را داخل یک بلاک *if* قرار می‌دهد. متعاقبا، نخ پس از دریافت اطلاع‌رسانی، از بررسی گزاره‌ی شرطی باز می‌ماند. اگر اطلاع‌رسانی جعلی یا مخرب باشد، نخ پیش از موعد مقرر بیدار می‌شود.

```
#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.
}
```

مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

### ۲,۵,۱۰ راه‌حل سازگار

این راه‌حل، تابع عضو *wait()* را داخل حلقه‌ی *while* فرا می‌خواند تا شرط را قبل و بعد از *wait()* بررسی کند.

```
#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);
	if (list.next == nullptr)
	{
		condition.wait(lk);
	}
	// Proceed when condition holds.
}
```

### ۳,۵,۱۰ راه‌حل سازگار (حلقه‌ی صریح با گزاره‌ی لاندا)

تابع *std::condition_variable::wait()*، دارای یک فرم سربار است که شی تابعی را، که نمایانگر گزاره است، می‌پذیرد. این فرم *wait()*، به‌گونه‌ای رفتار می‌کند که گویی به عنوان *while (!pred()) wait(lock);* پیاده‌سازی شده‌است. این راه‌حل، از یک لاندا به‌عنوان گزاره استفاده می‌کند و آن را به تابع *wait()* گذر می‌دهد. انتظار می‌رود زمانی‌که گزاره جهت اقدام، ایمن است، ارزش درست را بازگرداند. استفاده از گزاره‌ی حلقه‌ی صریح، منطق گزاره را در راه‌حل سازگار معکوس می‌کند.

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکزماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

```
#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.
}
```

### ۴,۵,۱۰ارزیابی خطر

عدم موفقیت در محصور کردن فراخوانی‌های توابع عضو *wait_for ()*، *wait ()*، و *wait_until* داخل حلقه، می‌تواند منجر به بلاک نامعین و انکار خدمات شود.

| قانون | شدت/دقت | احتمال | هزینه‌ی اصلاح | اولویت | سطح |
|---|---|---|---|---|---|
| **CON54-CPP** | پایین | بعید | متوسط | P2 | L3 |

مدیریت مرکز امداد و هماهنگی رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکزماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

## ۶.۱۰ CON55-CPP هنگام استفاده از متغیرهای شرطی، ایمنی و حیات نخ را حفظ کنید

ایمنی و حیات نخ‌ها از جمله معیارهای حیاتی در هنگام استفاده از متغیرهای شرطی به‌شمار می‌روند. ویژگی ایمنی نخ نیاز دارد تمامی حالات اشیا را در محیط چندنخی، به‌صورت ثابت حفظ نماید. بنابر ویژگی حیات، هر عملیات یا فراخوانی تابع جهت تکمیل بدون وقفه، اجرا می‌شوند. به‌عنوان مثال، هیچ بن‌بستی وجود ندارد.

باید متغیرهای شرطی باید داخل یک حلقه قرار گیرند. لازم است برنامه، جهت تضمین زنده بودن، شرط حلقه‌ی *while* را پیش از فراخوانی تابع عضو *condition_variable::wait()* بررسی کند. این تست اولیه، بررسی می‌نماید که آیا تاکنون نخ دیگری گزاره‌ی شرطی را راضی نموده و مبادرت به ارسال یک اعلان نموده‌است یا خیر. فراخوانی *wait()* پس از ارسال اعلان، منجر به بلاک نامعینی می‌شود.

برنامه باید برای تضمین ایمنی نخ، پس از بازگشت از *wait()*، شرط حلقه‌ی *while* را بیازماید. وقتی نخ *wait()* را فرا می‌خواند، سعی دارد تا زمانی‌که متغیر شرطی مربوطه توسط فراخوانی *condition_variable::notify_all()* یا *condition_variable::notify_one()* سیگنال داده می‌شود، بلاک کند.

تابع عضو *notify_one()* یکی از نخ‌هایی را که تحت متغیر شرطی خاص بلاک می‌شود، هنگام فراخوانی آزاد می‌کند. اگر چندین نخ منتظر یک متغیر شرطی باشند، زمان‌بند می‌تواند یکی از نخ‌ها را برای بیدار کردن انتخاب نماید (فرض بر آن است که سطح اولویت تمامی نخ‌ها یکسان است).

تابع عضو *notify_all()*، تمام نخ‌هایی را که تحت متغیر شرطی خاص بلاک می‌شوند، هنگام فراخوانی آزاد می‌کند. ترتیب اجرای نخ‌ها پس از فراخوانی *notify_all()* مشخص نیست. در نتیجه، یک نخ نامربوط می‌تواند اجرا را آغاز نماید. برطرف شدن گزاره‌ی شرطی را کشف کند، و حتی در صورتی‌که قرار باشد خوابیده بماند، اجرا را ادامه دهد.

بنابر دلایل مذکور، لازم است نخ‌ها گزاره‌ی شرطی را پس از بازگشت‌های تابع *wait()* بررسی نمایند. یک حلقه‌ی *while* بهترین انتخاب برای بررسی گزاره‌ی شرطی، پیش و پس از فراخوانی *wait()* است.

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

در صورتی‌که هر نخ از یک متغیر شرطی منحصر به‌فرد استفاده کند، استفاده از *notify_one()* امن خواهد بود. اگر چند نخ یک متغیر شرطی را به اشتراک بگذارند، استفاده از *notify_one()* تنها در صورتی امن خواهد بود که شرایط زیر برقرار باشد:

- تمام نخ‌ها باید یک مجموعه از عملیات را پس از بیدار شدن، انجام دهند. این بدان معنا است که هر نخ می‌تواند جهت بیدار شدن و از سرگیری فراخوانی *notify_one()*، انتخاب شود.
- تنها به بیدار بودن یک نخ جهت دریافت سیگنال نیاز است.

در صورتی‌که استفاده از *notify_one()* ناامن باشد، می‌توان از تابع *notify_all()* برای آزاد کردن تمام نخ‌هایی که روی یک متغیر شرطی مشخص بلاک شده‌اند، استفاده کرد.

### ۱.۶.۱۰ نمونه‌کد ناسازگار (*notify_one()*)

این مثال، از ۵ نخ استفاده می‌کند. نخ‌ها در راستای اجرای ترتیبی مبتنی بر سطح گام تخصیصی به هر نخ در زمان ایجاد (پردازش متوالی)، به کار گرفته می‌شوند. متغیر *TheCurrentStep* سطح گام فعلی را نگه می‌دارد. وقتی نخ مورد نظر تکمیل می‌شود، یک واحد به مقدار این متغیر افزوده می‌شود. سرانجام، نخ دیگری سیگنال داده می‌شود تا گام بعدی انجام پذیرد. هر نخ منتظر می‌ماند تا سطح گام مرتبط با آن آماده گردد و *wait()* نیز، داخل یک حلقه‌ی *while* پوشانده شود.

```
#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;
	}
```

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکزماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

```
        // 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();
        }
}
```

در این مثال، همه‌ی نخ‌ها یک متغیر شرطی را به اشتراک می‌گذراند. هر نخ، دارای گزاره‌ی شرطی متمایزی است، زیرا هر نخ به *currentStep* نیاز دارد تا پیش از اقدام، مقدار متفاوتی داشته باشد. وقتی به متغیر شرطی سیگنال داده می‌شود، هر نخ منتظری می‌تواند بیدار شود. جدول زیر سناریوهای ممکن را، که در آن ویژگی حیات نقض شده‌است، نشان می‌دهد. اگر با احتمالی، نخ اعلان‌شده همان نخ با مقدار گام بعدی نباشد، آن نخ دوباره منتظر می‌ماند. هیچ اعلان اضافه‌ای نمی‌تواند اتفاق بیفتد و در نهایت، استخر نخ‌ها خشک خواهد شد.

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

**بن بست: خارج از ترتیب مقدار گام**

| *Time* | *Thread # (my_step)* | *current_step* | *Action* |
|---|---|---|---|
| *0* | *3* | *0* | *Thread 3 executes the first time: the predicate is false -> wait()* |
| *1* | *2* | *0* | *Thread 2 executes the first time: the predicate is false -> wait()* |
| *2* | *4* | *0* | *Thread 4 executes the first time: the predicate is false -> wait()* |
| *3* | *0* | *0* | *Thread 0 executes the first time: the predicate is true -> currentStep++; notify_one()* |
| *4* | *1* | *1* | *Thread 1 executes the first time: the predicate is true -> currentStep++; notify_one()* |
| *5* | *3* | *2* | *Thread 3 wakes up (scheduler choice): the predicate is false -> wait()* |
| *6* | — | — | ***Thread exhaustion!** There are no more threads to run, and a conditional variable signal is needed to wake up the others.* |

### ۲،۶،۱۰ راه‌حل سازگار (*notify_all()*)

این راه‌حل، به جای سیگنال دادن به یک نخ تصادفی، از *notify_all()* برای سیگنال دادن به تمام نخ‌ها استفاده می‌کند. فقط کد نخ *run_step()* موجود در نمونه‌کد ناسازگار تغییر می‌یابد.

```
#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;
	}
```

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

```
        // 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 ...
```

بیدار کردن تمام نخ‌ها، ویژگی زنده‌بودن را تضمین می‌کند. زیرا هر نخ، بررسی گزاره‌ی شرطی مختص به خود را انجام می‌دهد و در این میان، تنها یکی موفق خواهد بود و به اجرا ادامه خواهد داد.

### ۳.۶.۱۰ راه‌حل سازگار (استفاده از *notify_one()* با یک متغیر شرطی منحصر به‌فرد برای هر نخ)

یک راه‌حل سازگار دیگر این است یک متغیر شرطی منحصر به‌فرد (و همه مرتبط با یک میوتکس) برای هر نخ استفاده شود. در این مورد، *notify_one()* فقط نخی را که منتظر آن است، بیدار می‌کند. این راه‌حل، از کارایی بیشتری در مقایسه با روش از استفاده از *notify_all()* برخوردار است. زیرا فقط نخ موردنظر بیدار می‌شود. باید گزاره‌ی شرطی نخ سیگنال داده‌شده درست باشد. در غیر این‌صورت، یک بن‌بست رخ خواهد داد.

```
#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;
        }
```

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکزماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

```
        // 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 ...
```

### ۴,۶,۱۰ارزیابی خطر

عدم موفقیت در حفظ ایمنی و حیات نخ به هنگام استفاده از یک متغیر شرطی، می‌تواند بلاک شدن نامعین یا انکار خدمات را در پی داشته‌باشد.

| قانون | شدت/دقت | احتمال | هزینه‌ی اصلاح | اولویت | سطح |
|---|---|---|---|---|---|
| CON55-CPP | پایین | بعید | متوسط | P2 | L3 |

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

## ۷,۱۰ CON56-CPP یک میوتکس غیربازگشتی را که متعلق به فراخوانی نخ است، قفل نکنید

کتابخانه‌ی استاندارد C++، کلاس‌های بازگشتی و غیر بازگشتی میوتکس را، که برای محافظت در مناطق بحرانی استفاده می‌شوند، تأمین می‌کند. کلاس‌های بازگشتی میوتکس (*std::recursive_mutex* و *std::recursive_timed_mutex*) با کلاس‌های غیربازگشتی آن (*std::mutex, std::timed_mutex* و *std::shared_timed_mutex*) از این نظر که ممکن است یک کلاس میوتکس بازگشتی، به‌طور بازگشتی توسط نخی که مالک میوتکس است قفل شود، تفاوت دارند. تمام کلاس‌های میوتکس از این قابلیت برخوردارند که میوتکس‌ها را به‌صورت نظری و از طریق توابعی مانند *try_lock()*، *try_lock_for()*، *try_lock_until()*، *try_lock_shared_for ()*، و *try_lock_shared_until()* قفل کنند. این توابع قفل نظری سعی دارند مالکیت میوتکس نخ فراخوان را در دست گیرند. اما در حالتی که نتوانند موفق به این کار شوند، قفل نمی‌کنند. در عوض، یک مقدار از نوع بولین[۷۷] را، که تعیین می‌کند مالکیت اخذ شده‌است یا خیر، برمی‌گردانند.

یک نخ تلاش دارد تا مالکیت یک *shared_timed_mutex* را به‌صورت بازگشتی، در دست گیرد. بنابراین، تلاش برای برای قفل کردن نظری یک شی میوتکس، که مالکیت آن در دست نخ فراخوان است، یک رفتار تعریف‌نشده به‌شمار می‌رود توابع *try_lock()*، *try_lock_for()*، *try_lock_until()*، *try_lock_shared_for ()*، و *try_lock_shared_until()* را توسط نخ صاحب شی میوتکس غیربازگشتی، روی آن شی فرا نخوانید.

### ۱,۷,۱۰ نمونه‌کد ناسازگار

در این مثال، میوتکس *m* توسط نقطه‌ی ورودی ابتدایی نخ قفل شده و در تابع *do_work()* نیز، به‌صورت نظری توسط همان نخ قفل شده‌است. این امر، رفتار تعریف‌نشده‌ای را نتیجه می‌دهد، زیرا میوتکس از نوع بازگشتی نیست. ممکن است کد مذکور، با پیاده‌سازی‌های معمول به بن‌بست ختم شود.

---

[۷۷] Boolean

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

```
#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();
}
```

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

### ۲,۷,۱۰ راه‌حل سازگار

این راه‌حل، قفل را از نقطه‌ی ورودی ابتدایی نخ حذف می‌کند و به میوتکس اجازه می‌دهد تا به‌صورت نظری قفل شود، نه به‌صورت بازگشتی.

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

```
#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();
}
```

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

### ۳.۷.۱۰ ارزیابی خطر

قفل کردن نظری یک میوتکس غیربازگشتی به روش بازگشتی، رفتار تعریف‌نشده‌ای است که می‌تواند منجر به بن‌بست شود.

| قانون | شدت/دقت | احتمال | هزینه‌ی اصلاح | اولویت | سطح |
|---|---|---|---|---|---|
| CON56-CPP | پایین | بعید | بالا | P1 | L3 |

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

# پیوست الف

# ارزیابی‌های خطر

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

| قانون | شدت/دقت | احتمال | هزینه‌ی اصلاح | اولویت | سطح |
|---|---|---|---|---|---|
| ERR54-CPP | متوسط | احتمال زیاد | پایین | P18 | L1 |
| MEM50-CPP | بالا | احتمال زیاد | متوسط | P18 | L1 |
| MEM51-CPP | بالا | احتمال زیاد | متوسط | P18 | L1 |
| MEM52-CPP | بالا | احتمال زیاد | متوسط | P18 | L1 |
| MEM53-CPP | بالا | احتمال زیاد | متوسط | P18 | L1 |
| MEM54-CPP | بالا | احتمال زیاد | متوسط | P18 | L1 |
| MEM55-CPP | بالا | احتمال زیاد | متوسط | P18 | L1 |
| MEM56-CPP | بالا | احتمال زیاد | متوسط | P18 | L1 |
| STR50-CPP | بالا | احتمال زیاد | متوسط | P18 | L1 |
| STR51-CPP | بالا | احتمال زیاد | متوسط | P18 | L1 |
| DCL50-CPP | بالا | محتمل | متوسط | P12 | L1 |
| ERR59-CPP | بالا | محتمل | متوسط | P12 | L1 |
| EXP53-CPP | بالا | محتمل | متوسط | P12 | L1 |
| EXP60-CPP | بالا | محتمل | متوسط | P12 | L1 |
| ERR55-CPP | پایین | احتمال زیاد | پایین | P9 | L2 |
| ERR56-CPP | بالا | احتمال زیاد | بالا | P9 | L2 |
| ERR58-CPP | پایین | احتمال زیاد | پایین | P9 | L2 |
| OOP52-CPP | پایین | احتمال زیاد | پایین | P9 | L2 |
| OOP58-CPP | پایین | احتمال زیاد | پایین | P9 | L2 |
| CON52-CPP | متوسط | محتمل | متوسط | P8 | L2 |
| EXP50-CPP | متوسط | محتمل | متوسط | P8 | L2 |
| EXP55-CPP | متوسط | محتمل | متوسط | P8 | L2 |
| EXP63-CPP | متوسط | محتمل | متوسط | P8 | L2 |
| CON51-CPP | پایین | محتمل | پایین | P6 | L2 |

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکزماهر
مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

| | | | | | |
|---|---|---|---|---|---|
| DCL54-CPP | پایین | محتمل | پایین | P6 | L2 |
| DCL58-CPP | بالا | بعید | متوسط | P6 | L2 |
| DCL57-CPP | پایین | احتمال زیاد | متوسط | P6 | L3 |
| EXP54-CPP | بالا | محتمل | بالا | P6 | L2 |
| EXP61-CPP | بالا | محتمل | بالا | P6 | L2 |
| EXP62-CPP | بالا | محتمل | بالا | P6 | L2 |
| FIO50-CPP | پایین | احتمال زیاد | متوسط | P6 | L2 |
| MEM57-CPP | متوسط | بعید | پایین | P6 | L2 |
| OOP55-CPP | بالا | محتمل | بالا | P6 | L2 |
| OOP57-CPP | بالا | محتمل | بالا | P6 | L2 |
| STR52-CPP | بالا | محتمل | بالا | P6 | L2 |
| STR53-CPP | بالا | بعید | متوسط | P6 | L2 |
| CON50-CPP | متوسط | محتمل | بالا | P4 | L3 |
| CON53-CPP | پایین | محتمل | متوسط | P4 | L3 |
| ERR50-CPP | پایین | محتمل | متوسط | P4 | L3 |
| ERR51-CPP | پایین | محتمل | متوسط | P4 | L3 |
| ERR52-CPP | پایین | محتمل | متوسط | P4 | L3 |
| ERR60-CPP | پایین | محتمل | متوسط | P4 | L3 |
| EXP57-CPP | متوسط | بعید | متوسط | P4 | L3 |
| EXP58-CPP | متوسط | بعید | متوسط | P4 | L3 |
| EXP59-CPP | متوسط | بعید | متوسط | P4 | L3 |
| ERR62-CPP | متوسط | بعید | متوسط | P4 | L3 |
| FIO51-CPP | متوسط | بعید | متوسط | P4 | L3 |
| INT50-CPP | متوسط | بعید | متوسط | P4 | L3 |
| OOP51-CPP | پایین | محتمل | متوسط | P4 | L3 |

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران
مرکزماهر
مرکز مدیریت امداد و هماهنگی عملیات رخدادهای رایانه ای

| | | | | | |
|---|---|---|---|---|---|
| OOP53-CPP | متوسط | بعید | متوسط | P4 | L3 |
| DCL59-CPP | پایین | بعید | متوسط | P4 | L3 |
| DCL60-CPP | بالا | بعید | بالا | P3 | L3 |
| DCL51-CPP | پایین | بعید | پایین | P3 | L3 |
| DCL52-CPP | پایین | بعید | پایین | P3 | L3 |
| ERR61-CPP | پایین | بعید | پایین | P3 | L3 |
| EXP52-CPP | پایین | بعید | پایین | P3 | L3 |
| CON54-CPP | پایین | بعید | متوسط | P2 | L3 |
| CON55-CPP | پایین | بعید | متوسط | P2 | L3 |
| DCL53-CPP | پایین | بعید | متوسط | P2 | L3 |
| DCL56-CPP | پایین | بعید | متوسط | P2 | L3 |
| EXP56-CPP | پایین | بعید | متوسط | P2 | L3 |
| ERR53-CPP | پایین | بعید | متوسط | P2 | L3 |
| ERR57-CPP | پایین | محتمل | بالا | P2 | L3 |
| EXP51-CPP | پایین | بعید | متوسط | P2 | L3 |
| OOP50-CPP | پایین | بعید | متوسط | P2 | L3 |
| OOP54-CPP | پایین | محتمل | بالا | P2 | L3 |
| OOP56-CPP | پایین | محتمل | بالا | P2 | L3 |
| CON56-CPP | پایین | بعید | بالا | P1 | L3 |
| DCL55-CPP | پایین | بعید | بالا | P1 | L3 |

وزارت ارتباطات و فناوری اطلاعات
سازمان فناوری اطلاعات ایران

مرکز ماهر
مرکز مدیریت امداد و هماهنگی
عملیات رخدادهای رایانه ای

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای

# مراجع

[1] Ballman, “SEI CERT C++ Coding Standard: Rules for Developing Safe, Reliable, and Secure System in C++,” Carnegie Mellon University, 2016.
[2] Wikipedia, https://www.wikipedia.org/.

مدیریت مرکز امداد و هماهنگی عملیات رخدادهای رایانه ای