DCL57-CPP. Do not let exceptions escape from destructors or deallocation functions

Under certain circumstances, terminating a destructor, `operator delete`, or `operator delete[]` by throwing an exception can trigger *undefined behavior*.

For instance, the C++ Standard, [basic.stc.dynamic.deallocation], paragraph 3 [ISO/IEC 14882-2014], in part, states the following:

> If a deallocation function terminates by throwing an exception, the behavior is undefined.

In these situations, the function must logically be declared `noexcept` because throwing an exception from the function can never have well-defined behavior. The C++ Standard, [except.spec], paragraph 15, states the following:

> A deallocation function with no explicit exception-specification is treated as if it were specified with `noexcept(true)`.

As such, deallocation functions (object, array, and placement forms at either global or class scope) must not terminate by throwing an exception. Do not declare such functions to be `noexcept(false)`. However, it is acceptable to rely on the implicit `noexcept(true)` specification or declare `noexcept` explicitly on the function signature.

Object destructors are likely to be called during stack unwinding as a result of an exception being thrown. If the destructor itself throws an exception, having been called as the result of an exception being thrown, then the function `std::terminate()` is called with the default effect of calling `std::abort()` [ISO/IEC 14882-2014]. When `std::abort()` is called, no further objects are destroyed, resulting in an indeterminate program state and undefined behavior. Do not terminate a destructor by throwing an exception.

The C++ Standard, [class.dtor], paragraph 3, states [ISO/IEC 14882-2014] the following:

> A declaration of a destructor that does not have an exception-specification is implicitly considered to have the same exception-specification as an implicit declaration.

An implicit declaration of a destructor is considered to be `noexcept(true)` according to [except.spec], paragraph 14. As such, destructors must not be declared `noexcept(false)` but may instead rely on the implicit `noexcept(true)` or declare `noexcept` explicitly.

Any `noexcept` function that terminates by throwing an exception violates ERR55-CPP. Honor exception specifications.

**Noncompliant Code Example**

In this noncompliant code example, the class destructor does not meet the implicit `noexcept` guarantee because it may throw an exception even if it was called as the result of an exception being thrown. Consequently, it is declared as `noexcept(false)` but still can trigger *undefined behavior*.

```cpp
#include <stdexcept>

class S {
    bool has_error() const;

public:
    ~S() noexcept(false) {
        // Normal processing
        if (has_error()) {
            throw std::logic_error("Something bad");
        }
    }
};
```

**Noncompliant Code Example (`std::uncaught_exception()`)**

Use of `std::uncaught_exception()` in the destructor solves the termination problem by avoiding the propagation of the exception if an existing exception is being processed, as demonstrated in this noncompliant code example. However, by circumventing normal destructor processing, this approach may keep the destructor from releasing important resources.
```cpp
#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");
        }
    }
};
```

**Noncompliant Code Example (function-try-block)**

This noncompliant code example, as well as the following compliant solution, presumes the existence of a `Bad` class with a destructor that can throw. Although the class violates this rule, it is presumed that the class cannot be modified to comply with this rule.

```cpp
// 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);
};
```

To safely use the `Bad` class, the `SomeClass` destructor attempts to handle exceptions thrown from the `Bad` destructor by absorbing them.

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

However, the C++ Standard, [except.handle], paragraph 15 [ISO/IEC 14882-2014], in part, states the following:

> The currently handled exception is rethrown if control reaches the end of a handler of the function-try-block of a constructor or destructor.

Consequently, the caught exception will inevitably escape from the `SomeClass` destructor because it is implicitly rethrown when control reaches the end of the `function-try-block` handler.

**Compliant Solution**

A destructor should perform the same way whether or not there is an active exception. Typically, this means that it should invoke only operations that do not throw exceptions, or it should handle all exceptions and not rethrow them (even implicitly). This compliant solution differs from the previous noncompliant code example by having an explicit `return` statement in the `SomeClass` destructor. This statement prevents control from reaching the end of the exception handler. Consequently, this handler will catch the exception thrown by `Bad::~Bad()` when `bad_member` is destroyed. It will also catch any exceptions thrown within the compound statement of the `function-try-block`, but the `SomeClass` destructor will not terminate by throwing an exception.
```cpp
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;
  }
};
```

**Noncompliant Code Example**

In this noncompliant code example, a global deallocation is declared `noexcept(false)` and throws an exception if some conditions are not properly met. However, throwing from a deallocation function results in undefined behavior.

```cpp
#include <stdexcept>

bool perform_dealloc(void *);

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

**Compliant Solution**

The compliant solution does not throw exceptions in the event the deallocation fails but instead fails as gracefully as possible.

```cpp
#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
  }
}
```

**Risk Assessment**

Attempting to throw exceptions from destructors or deallocation functions can result in undefined behavior, leading to resource leaks or denial-of-service attacks.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Severity</th>
<th>Likelihood</th>
<th>Remediation Cost</th>
<th>Priority</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCL57-CPP</td>
<td>Low</td>
<td>Likely</td>
<td>Medium</td>
<td>P6</td>
<td>L2</td>
</tr>
</tbody>
</table>

**Automated Detection**

<table>
<thead>
<tr>
<th>Tool</th>
<th>Version</th>
<th>Checker</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tool</td>
<td>Version</td>
<td>Rule</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------</td>
<td>---------</td>
<td>------------------------------------</td>
<td>-------------------------------------------------</td>
</tr>
<tr>
<td>Astrée</td>
<td>22.10</td>
<td>destructor-without-noexcept</td>
<td>Fully checked</td>
</tr>
<tr>
<td></td>
<td></td>
<td>delete-without-noexcept</td>
<td></td>
</tr>
<tr>
<td>Axivion Bauhaus Suite</td>
<td>7.2.0</td>
<td>CertC++-DCL57</td>
<td></td>
</tr>
<tr>
<td>CodeSonar</td>
<td>8.0p0</td>
<td>LANG.STRUCT.EXCP.CATCH</td>
<td>Use of catch</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LANG.STRUCT.EXCP.THROW</td>
<td>Use of throw</td>
</tr>
<tr>
<td>Helix QAC</td>
<td>2023.3</td>
<td>C++2045, C++2047, C++4032, C++4631</td>
<td></td>
</tr>
<tr>
<td>Klocwork</td>
<td>2023.3</td>
<td>MISRA.DTOR.THROW</td>
<td></td>
</tr>
<tr>
<td>LDRA tool suite</td>
<td>9.7.1</td>
<td>453 S</td>
<td>Partially implemented</td>
</tr>
<tr>
<td>Parasoft C/C++test</td>
<td>2023.1</td>
<td>CERT_CPP-DCL57-a</td>
<td>Never allow an exception to be thrown from a destructor, deallocation, and swap</td>
</tr>
<tr>
<td>Polyspace Bug Finder</td>
<td>R2023b</td>
<td>CERT C++: DCL57-CPP</td>
<td>Checks for class destructors exiting with an exception (rule partially covered)</td>
</tr>
<tr>
<td>PVS-Studio</td>
<td>7.29</td>
<td>V509, V1045</td>
<td></td>
</tr>
<tr>
<td>RuleChecker</td>
<td>22.10</td>
<td>destructor-without-noexcept</td>
<td>Fully checked</td>
</tr>
<tr>
<td></td>
<td></td>
<td>delete-without-noexcept</td>
<td></td>
</tr>
</tbody>
</table>

**Related Vulnerabilities**

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

**Related Guidelines**

<table>
<thead>
<tr>
<th>Reference</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEI CERT C++ Coding Standard</td>
<td>ERR55-CPP. Honor exception specifications</td>
</tr>
<tr>
<td></td>
<td>ERR50-CPP. Do not abruptly terminate the program</td>
</tr>
<tr>
<td>MISRA C++:2008</td>
<td>Rule 15-5-1 (Required)</td>
</tr>
</tbody>
</table>

**Bibliography**

[Henricson 1997] Recommendation 12.5, Do not let destructors called during stack unwinding throw exceptions

Subclause 15.2, “Constructors and Destructors”
Subclause 15.3, “Handling an Exception”
Subclause 15.4, “Exception Specifications”

[Meyers 2005] Item 8, “Prevent Exceptions from Leaving Destructors”

[Sutter 2000] “Never allow exceptions from escaping destructors or from an overloaded operator delete()” (p. 29)