ERR60-CPP. Exception objects must be nothrow copy constructible

When an exception is thrown, the exception object operand of the throw expression is copied into a temporary object that is used to initialize the handler. The C++ Standard, [except.throw], paragraph 3 [ISO/IEC 14882-2014], in part, states the following:

Throwing an exception copy-initializes a temporary object, called the exception object. The temporary is an lvalue and is used to initialize the variable declared in the matching handler.

If the copy constructor for the exception object type throws during the copy initialization, std::terminate() is called, which can result in possibly unexpected implementation-defined behavior. For more information on implicitly calling std::terminate(), see ERR50-CPP. Do not abruptly terminate the program.

The copy constructor for an object thrown as an exception must be declared noexcept, including any implicitly-defined copy constructors. Any function declared noexcept that terminates by throwing an exception violates ERR55-CPP. Honor exception specifications.

The C++ Standard allows the copy constructor to be elided when initializing the exception object to perform the initialization if a temporary is thrown. Many modern compiler implementations make use of both optimization techniques. However, the copy constructor for an exception object still must not throw an exception because compilers are not required to elide the copy constructor call in all situations, and common implementations of std::exception_ptr will call a copy constructor even if it can be elided from a throw expression.

Noncompliant Code Example

In this noncompliant code example, an exception of type S is thrown in f(). However, because S has a std::string data member, and the copy constructor for std::string is not declared noexcept, the implicitly-defined copy constructor for S is also not declared to be noexcept. In low-memory situations, the copy constructor for std::string may be unable to allocate sufficient memory to complete the copy operation, resulting in a std::bad_alloc exception being thrown.

```cpp
#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
   }
}
```

Compliant Solution

This compliant solution assumes that the type of the exception object can inherit from std::runtime_error, or that type can be used directly. Unlike std::string, a std::runtime_error object is required to correctly handle an arbitrary-length error message that is exception safe and guarantees the copy constructor will not throw [ISO/IEC 14882-2014].
#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
    }
}

Compliant Solution

If the exception type cannot be modified to inherit from std::runtime_error, a data member of that type is a legitimate implementation strategy, as shown in this compliant solution.

#include <stdexcept>
#include <type_traits>

class S : public std::exception {
    std::runtime_error m;
public:
    S(const char *msg) : m(msg) {}
    const char *what() const noexcept override {
        return m.what();
    }
};

static_assert(std::is_nothrow_copy_constructible<S>::value,
              "S must be nothrow copy constructible");

void g() {
    // If some condition doesn't hold...
    throw S("Condition did not hold");
}

void f() {
    try {
        g();
    } catch (S &s) {
        // Handle error
    }
}

Risk Assessment

Allowing the application to abnormally terminate can lead to resources not being freed, closed, and so on. It is frequently a vector for 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>ERR60-CPP</td>
<td>Low</td>
<td>Probable</td>
<td>Medium</td>
<td>P4</td>
<td>L3</td>
</tr>
</tbody>
</table>
# Automated Detection Tool Version Checker Description

<table>
<thead>
<tr>
<th>Tool</th>
<th>Version</th>
<th>Checker</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clang</td>
<td>3.9</td>
<td>cert-err60-cpp</td>
<td>Checked by clang-tidy</td>
</tr>
<tr>
<td>Helix QAC</td>
<td>2021.2</td>
<td>C++3508</td>
<td></td>
</tr>
<tr>
<td>Parasoft C/C++test</td>
<td>2021.2</td>
<td>CERT_CPP-ERR60-a CERT_CPP-ERR60-b</td>
<td>Exception objects must be nothrow copy constructible An explicitly declared copy constructor for a class that inherits from 'std::exception' should have a non-throwing exception specification</td>
</tr>
<tr>
<td>PRQA QA-C++</td>
<td>4.4</td>
<td>3508</td>
<td></td>
</tr>
</tbody>
</table>

## Related Vulnerabilities

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

## Related Guidelines

<table>
<thead>
<tr>
<th>SEI CERT C++ Coding Standard</th>
<th>ERR50-CPP. Do not abruptly terminate the program</th>
<th>ERR55-CPP. Honor exception specifications</th>
</tr>
</thead>
</table>

## Bibliography

- [Hinnant 2015](#)