ERR50-CPP. Do not abruptly terminate the program

The `std::abort()`, `std::quick_exit()`, and `std::_Exit()` functions are used to terminate the program in an immediate fashion. They do so without calling exit handlers registered with `std::atexit()` and without executing destructors for objects with automatic, thread, or static storage duration. How a system manages open streams when a program ends is implementation-defined [ISO/IEC 9899:1999]. Open streams with unwritten buffered data may or may not be flushed, open streams may or may not be closed, and temporary files may or may not be removed. Because these functions can leave external resources, such as files and network communications, in an indeterminate state, they should be called explicitly only in direct response to a critical error in the application. (See ERR50-CPP-EX1 for more information.)

The `std::terminate()` function calls the current `terminate_handler` function, which defaults to calling `std::abort()`.

The C++ Standard defines several ways in which `std::terminate()` may be called implicitly by an implementation [ISO/IEC 14882-2014]:

1. When the exception handling mechanism, after completing the initialization of the exception object but before activation of a handler for the exception, calls a function that exits via an exception ([except.throw], paragraph 7)
    - See ERR60-CPP. Exception objects must be nothrow copy constructible for more information.
2. When a *throw-expression* with no operand attempts to rethrow an exception and no exception is being handled ([except.throw], paragraph 9)
3. When the exception handling mechanism cannot find a handler for a thrown exception ([except.handle], paragraph 9)
    - See ERR51-CPP. Handle all exceptions for more information.
4. When the search for a handler encounters the outermost block of a function with a *noexcept-specification* that does not allow the exception ([except.spec], paragraph 9)
    - See ERR55-CPP. Honor exception specifications for more information.
5. When the destruction of an object during stack unwinding terminates by throwing an exception ([except.ctor], paragraph 3)
    - See DCL57-CPP. Do not let exceptions escape from destructors or deallocation functions for more information.
6. When initialization of a nonlocal variable with static or thread storage duration exits via an exception ([basic.start.init], paragraph 6)
    - See ERR58-CPP. Handle all exceptions thrown before main() begins executing for more information.
7. When destruction of an object with static or thread storage duration exits via an exception ([basic.start.term], paragraph 1)
    - See DCL57-CPP. Do not let exceptions escape from destructors or deallocation functions for more information.
8. When execution of a function registered with `std::atexit()`or `std::at_quick_exit()` exits via an exception ([support.start.term], paragraphs 8 and 12)
9. When the implementation's default unexpected exception handler is called ([except.unexpected], paragraph 2)
   Note that `std::unexpected()` is currently deprecated.
10. When `std::unexpected()` throws an exception that is not allowed by the previously violated *dynamic-exception-specification*, and `std::bad_exception()` is not included in that *dynamic-exception-specification* ([except.unexpected], paragraph 3)
11. When the function `std::nested_exception::rethrow_nested()` is called for an object that has captured no exception ([except.nested], paragraph 4)
12. When execution of the initial function of a thread exits via an exception ([thread.thread.constr], paragraph 5)
    - See ERR51-CPP. Handle all exceptions for more information.
13. When the destructor is invoked on an object of type `std::thread` that refers to a joinable thread ([thread.thread.destr], paragraph 1)
14. When the copy assignment operator is invoked on an object of type `std::thread` that refers to a joinable thread ([thread.thread.assign], paragraph 1)
15. When calling `condition_variable::wait()`, `condition_variable::wait_until()`, or `condition_variable::wait_for()` results in a failure to meet the postcondition: `lock.owns_lock() == true` or `lock.mutex()` is not locked by the calling thread ([thread.condition.condvar], paragraphs 11, 16, 21, 28, 33, and 40)
16. When calling `condition_variable_any::wait()`, `condition_variable_any::wait_until()`, or `condition_variable_any::wait_for()` results in a failure to meet the postcondition: `lock` is not locked by the calling thread ([thread.condition.condvarany], paragraphs 11, 16, and 22)

In many circumstances, the call stack will not be unwound in response to an implicit call to `std::terminate()`, and in a few cases, it is implementation-defined whether or not stack unwinding will occur. The C++ Standard, [except.terminate], paragraph 2 [ISO/IEC 14882-2014], in part, states the following:

> *In the situation where no matching handler is found, it is implementation-defined whether or not the stack is unwound before `std::terminate()` is called. In the situation where the search for a handler encounters the outermost block of a function with a* noexcept *-specification that does not allow the exception, it is implementation-defined whether the stack is unwound, unwound partially, or not unwound at all before `std::terminate()` is called. In all other situations, the stack shall not be unwound before `std::terminate()` is called.*

Do not explicitly or implicitly call `std::quick_exit()`, `std::abort()`, or `std::_Exit()`. When the default `terminate_handler` is installed or the current `terminate_handler` responds by calling `std::abort()` or `std::_Exit()`, do not explicitly or implicitly call `std::terminate()`. Abnormal process termination is the typical vector for denial-of-service attacks.

It is acceptable to call a termination function that safely executes destructors and properly cleans up resources, such as `std::exit()`.

Noncompliant Code Example

In this noncompliant code example, the call to `f()`, which was registered as an exit handler with `std::at_exit()`, may result in a call to `std::terminate()` because `throwing_func()` may throw an exception.

```
#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
  }
  // ...
}
```

Compliant Solution

In this compliant solution, `f()` handles all exceptions thrown by `throwing_func()` and does not rethrow.

```
#include <cstdlib>

void throwing_func() noexcept(false);

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

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

Exceptions

ERR50-CPP-EX1: It is acceptable, after indicating the nature of the problem to the operator, to explicitly call `std::abort()`, `std::_Exit()`, or `std::terminate()` in response to a critical program error for which no recovery is possible, as in this example.

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

The `assert()` macro is permissible under this exception because failed assertions will notify the operator on the standard error stream in an implementation-defined manner before calling `std::abort()`.

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.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
ERR50-CPP	Low	Probable	Medium	**P4**	**L3**

Automated Detection

Tool	Version	Checker	Description
CodeSonar	5.2p0	**BADFUNC.ABORT** **BADFUNC.EXIT**	Use of abort Use of exit
Klocwork	2018	**MISRA.CATCH.ALL**	
LDRA tool suite	9.7.1	**122 S**	Enhanced Enforcement
Parasoft C/C++test	10.4.2	**CERT_CPP-ERR50-a** **CERT_CPP-ERR50-b** **CERT_CPP-ERR50-c** **CERT_CPP-ERR50-d** **CERT_CPP-ERR50-e** **CERT_CPP-ERR50-f** **CERT_CPP-ERR50-g** **CERT_CPP-ERR50-h** **CERT_CPP-ERR50-i** **CERT_CPP-ERR50-j** **CERT_CPP-ERR50-k** **CERT_CPP-ERR50-l** **CERT_CPP-ERR50-m**	The execution of a function registered with 'std::atexit()' or 'std::at_quick_exit()' should not exit via an exception Never allow an exception to be thrown from a destructor, deallocation, and swap Do not throw from within destructor There should be at least one exception handler to catch all otherwise unhandled exceptions An empty throw (throw;) shall only be used in the compound-statement of a catch handler Exceptions shall be raised only after start-up and before termination of the program Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point Where a function's declaration includes an exception-specification, the function shall only be capable of throwing exceptions of the indicated type(s) Function called in global or namespace scope shall not throw unhandled exceptions Always catch exceptions Properly define exit handlers The library functions 'abort()', 'quick_exit()' and '_Exit()' from 'cstdlib' library shall not be used Avoid throwing exceptions from functions that are declared not to throw
Polyspace Bug Finder	R2019b	CERT C++: ERR50-CPP	Checks for implicit call to terminate() function (rule partially covered)
PRQA QA-C++	4.4	**5014**	
SonarQube C/C++ Plugin	4.10	**S990**	

Related Vulnerabilities

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

Related Guidelines

SEI CERT C++ Coding Standard	ERR51-CPP. Handle all exceptions ERR55-CPP. Honor exception specifications DCL57-CPP. Do not let exceptions escape from destructors or deallocation functions
MITRE CWE	CWE-754, Improper Check for Unusual or Exceptional Conditions

Bibliography

[ISO/IEC 9899-2011]	Subclause 7.20.4.1, "The `abort` Function" Subclause 7.20.4.4, "The `_Exit` Function"
[ISO/IEC 14882-2014]	Subclause 15.5.1, "The `std::terminate()` Function" Subclause 18.5, "Start and Termination"
[MISRA 2008]	Rule 15-3-2 (Advisory) Rule 15-3-4 (Required)