2 Declarations and Initialization (DCL)

2.1 DCL50-CPP. Do not define a C-style variadic function

Functions can be defined to accept more formal arguments at the call site than are specified by the parameter declaration clause. Such functions are called *variadic* functions because they can accept a variable number of arguments from a caller. C++ provides two mechanisms by which a variadic function can be defined: function parameter packs and use of a C-style ellipsis as the final parameter declaration.

Variadic functions are flexible because they accept a varying number of arguments of differing types. However, they can also be hazardous. A variadic function using a C-style ellipsis (hereafter called a *C-style variadic function*) has no mechanisms to check the type safety of arguments being passed to the function or to check that the number of arguments being passed matches the semantics of the function definition. Consequently, a runtime call to a C-style variadic function that passes inappropriate arguments yields undefined behavior. Such undefined behavior could be exploited to run arbitrary code.

Do not define C-style variadic functions. (The declaration of a C-style variadic function that is never defined is permitted, as it is not harmful and can be useful in unevaluated contexts.)

Issues with C-style variadic functions can be avoided by using variadic functions defined with function parameter packs for situations in which a variable number of arguments should be passed to a function. Additionally, function currying can be used as a replacement to variadic functions. For example, in contrast to C's `printf()` family of functions, C++ output is implemented with the overloaded single-argument `std::cout::operator<<()` operators.

2.1.1 Noncompliant Code Example

This noncompliant code example uses a C-style variadic function to add a series of integers together. The function reads arguments until the value `0` is found. Calling this function without passing the value `0` as an argument (after the first two arguments) results in undefined behavior. Furthermore, passing any type other than an `int` also results in undefined behavior.

```
#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;
}
```

2.1.2 Compliant Solution (Recursive Pack Expansion)

In this compliant solution, a variadic function using a function parameter pack is used to implement the `add()` function, allowing identical behavior for call sites. Unlike the C-style variadic function used in the noncompliant code example, this compliant solution does not result in undefined behavior if the list of parameters is not terminated with `0`. Additionally, if any of the values passed to the function are not integers, the code is ill-formed rather than producing undefined behavior.

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

This compliant solution makes use of `std::enable_if` to ensure that any nonintegral argument value results in an ill-formed program.

2.1.3 Compliant Solution (Braced Initializer List Expansion)

An alternative compliant solution that does not require recursive expansion of the function parameter pack instead expands the function parameter pack into a list of values as part of a braced initializer list. Since narrowing conversions are not allowed in a braced initializer list, the type safety is preserved despite the `std::enable_if` not involving any of the variadic arguments.

```
#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;
}
```

2.1.4 Exceptions

DCL50-CPP-EX1: It is permissible to define a C-style variadic function if that function also has external C language linkage. For instance, the function may be a definition used in a C library API that is implemented in C++.

DCL50-CPP-EX2: As stated in the normative text, C-style variadic functions that are declared but never defined are permitted. For example, when a function call expression appears in an unevaluated context, such as the argument in a `sizeof` expression, overload resolution is performed to determine the result type of the call but does not require a function definition. Some template metaprogramming techniques that employ SFINAE use variadic function declarations to implement compile-time type queries, as in the following example.

```
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);
};
```

In this example, the value of `value` is determined on the basis of which overload of `test()` is selected. The declaration of `Inner *I` allows use of the variable `I` within the `decltype` specifier, which results in a pointer of some (possibly `void`) type, with a default value of `nullptr`. However, if there is no declaration of `Inner::foo()`, the `decltype` specifier will be ill-formed, and that variant of `test()` will not be a candidate function for overload resolution due to SFINAE. The result is that the C-style variadic function variant of `test()` will be the only function in the candidate set. Both `test()` functions are declared but never defined because their definitions are not required for use within an unevaluated expression context.

2.1.5 Risk Assessment

Incorrectly using a variadic function can result in abnormal program termination, unintended information disclosure, or execution of arbitrary code.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
DCL50-CPP	High	Probable	Medium	**P12**	**L1**

2.1.6 Bibliography

[ISO/IEC 14882-2014]	Subclause 5.2.2, “Function Call” Subclause 14.5.3, “Variadic Templates”

2.2 DCL51-CPP. Do not declare or define a reserved identifier

The C++ Standard, [reserved.names] [ISO/IEC 14882-2014], specifies the following rules regarding reserved names:

> - A translation unit that includes a standard library header shall not `#define` or `#undef` names declared in any standard library header.
> - A translation unit shall not `#define` or `#undef` names lexically identical to keywords, to the identifiers listed in Table 3, or to the *attribute-token*s described in 7.6.
> - Each name that contains a double underscore __ or begins with an underscore followed by an uppercase letter is reserved to the implementation for any use.
> - Each name that begins with an underscore is reserved to the implementation for use as a name in the global namespace.
> - Each name declared as an object with external linkage in a header is reserved to the implementation to designate that library object with external linkage, both in namespace `std` and in the global namespace.
> - Each global function signature declared with external linkage in a header is reserved to the implementation to designate that function signature with external linkage.
> - Each name from the Standard C library declared with external linkage is reserved to the implementation for use as a name with `extern “C”` linkage, both in namespace `std` and in the global namespace.
> - Each function signature from the Standard C library declared with external linkage is reserved to the implementation for use as a function signature with both `extern “C”` and `extern “C++”` linkage, or as a name of namespace scope in the global namespace.
> - For each type T from the Standard C library, the types `::T` and `std::T` are reserved to the implementation and, when defined, `::T` shall be identical to `std::T`.
> - Literal suffix identifiers that do not start with an underscore are reserved for future standardization.

The identifiers and attribute names referred to in the preceding excerpt are `override`, `final`, `alignas`, `carries_dependency`, `deprecated`, and `noreturn`.

No other identifiers are reserved. Declaring or defining an identifier in a context in which it is reserved results in undefined behavior. Do not declare or define a reserved identifier.

2.2.1 Noncompliant Code Example (Header Guard)

A common practice is to use a macro in a preprocessor conditional that guards against multiple inclusions of a header file. While this is a recommended practice, many programs use reserved names as the header guards. Such a name may clash with reserved names defined by the implementation of the C++ standard template library in its headers or with reserved names implicitly predefined by the compiler even when no C++ standard library header is included.

```
#ifndef _MY_HEADER_H_
#define _MY_HEADER_H_

// Contents of <my_header.h>

#endif // _MY_HEADER_H_
```

2.2.2 Compliant Solution (Header Guard)

This compliant solution avoids using leading or trailing underscores in the name of the header guard.

```
#ifndef MY_HEADER_H
#define MY_HEADER_H

// Contents of <my_header.h>

#endif // MY_HEADER_H
```

2.2.3 Noncompliant Code Example (User-Defined Literal)

In this noncompliant code example, a user-defined literal operator"" x is declared. However, literal suffix identifiers are required to start with an underscore; literal suffixes without the underscore prefix are reserved for future library implementations.

```
#include <cstddef>

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

2.2.4 Compliant Solution (User-Defined Literal)

In this compliant solution, the user-defined literal is named `operator"" _x`, which is not a reserved identifier.

```
#include <cstddef>

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

The name of the user-defined literal is `operator"" _x` and not `_x`, which would have otherwise been reserved for the global namespace.

2.2.5 Noncompliant Code Example (File Scope Objects)

In this noncompliant code example, the names of the file scope objects `_max_limit` and `_limit` both begin with an underscore. Because it is `static`, the declaration of `_max_limit` might seem to be impervious to clashes with names defined by the implementation. However, because the header `<cstddef>` is included to define `std::size_t`, a potential for a name clash exists. (Note, however, that a conforming compiler may implicitly declare reserved names regardless of whether any C++ standard template library header has been explicitly included.) In addition, because `_limit` has external linkage, it may clash with a symbol with the same name defined in the language runtime library even if such a symbol is not declared in any header. Consequently, it is unsafe to start the name of any file scope identifier with an underscore even if its linkage limits its visibility to a single translation unit.

```
#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;
}
```

2.2.6 Compliant Solution (File Scope Objects)

In this compliant solution, file scope identifiers do not begin with an underscore.

```
#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;
}
```

2.2.7 Noncompliant Code Example (Reserved Macros)

In this noncompliant code example, because the C++ standard template library header `<cinttypes>` is specified to include `<cstdint>`, as per [c.files] paragraph 4 [ISO/IEC 14882-2014], the name `MAX_SIZE` conflicts with the name of the `<cstdint>` header macro used to denote the upper limit of `std:size_t`.

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

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

2.2.8 Compliant Solution (Reserved Macros)

This compliant solution avoids redefining reserved names.

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

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

2.2.9 Exceptions

DCL51-CPP-EX1: For compatibility with other compiler vendors or language standard modes, it is acceptable to create a macro identifier that is the same as a reserved identifier so long as the behavior is semantically identical, as in this example.

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

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

DCL51-CPP-EX2: As a compiler vendor or standard library developer, it is acceptable to use identifiers reserved for your implementation. Reserved identifiers may be defined by the compiler, in standard library headers, or in headers included by a standard library header, as in this example declaration from the libc++ STL implementation.

```
// 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;
};
```

2.2.10 Risk Assessment

Using reserved identifiers can lead to incorrect program operation.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
DCL51-CPP	Low	Unlikely	Low	**P3**	**L3**

2.2.11 Related Guidelines

SEI CERT C++ Coding Standard	DCL58-CPP. Do not modify the standard namespaces
SEI CERT C Coding Standard	DCL37-C. Do not declare or define a reserved identifier PRE06-C. Enclose header files in an inclusion guard
MISRA C++:2008	Rule 17-0-1

2.2.12 Bibliography

[ISO/IEC 14882-2014]	Subclause 17.6.4.3, "Reserved Names"
[ISO/IEC 9899:2011]	Subclause 7.1.3, "Reserved Identifiers"

2.3 DCL52-CPP. Never qualify a reference type with const or volatile

C++ does not allow you to change the value of a reference type, effectively treating all references as being `const` qualified. The C++ Standard, [dcl.ref], paragraph 1 [ISO/IEC 14882-2014], states the following:

> Cv-qualified references are ill-formed except when the cv-qualifiers are introduced through the use of a *typedef-name* (7.1.3, 14.1) or *decltype-specifier* (7.1.6.2), in which case the cv-qualifiers are ignored.

Thus, C++ prohibits or ignores the cv-qualification of a reference type. Only a value of non-reference type may be cv-qualified.

When attempting to `const`-qualify a type as part of a declaration that uses reference type, a programmer may accidentally write

```
char &const p;
```

instead of

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

Do not attempt to cv-qualify a reference type because it results in undefined behavior. A conforming compiler is required to issue a diagnostic message. However, if the compiler does not emit a fatal diagnostic, the program may produce surprising results, such as allowing the character referenced by `p` to be mutated.

2.3.1 Noncompliant Code Example

In this noncompliant code example, a `const`-qualified reference to a `char` is formed instead of a reference to a `const`-qualified `char.` This results in undefined behavior.

```
#include <iostream>

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

2.3.1.1 Implementation Details (MSVC)

With Microsoft Visual Studio 2015, this code compiles successfully with a warning diagnostic.

```
warning C4227: anachronism used : qualifiers on reference are ig-
nored
```

When run, the code outputs the following.

```
p
```

2.3.1.2 Implementation Details (Clang)

With Clang 3.9, this code produces a fatal diagnostic.

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

2.3.2 Noncompliant Code Example

This noncompliant code example correctly declares `p` to be a reference to a const-qualified `char`. The subsequent modification of `p` makes the program ill-formed.

```
#include <iostream>

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

2.3.3 Compliant Solution

This compliant solution removes the `const` qualifier.

```
#include <iostream>

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

2.3.4 Risk Assessment

A `const` or `volatile` reference type may result in undefined behavior instead of a fatal diagnostic, causing unexpected values to be stored and leading to possible data integrity violations.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
DCL52-CPP	Low	Unlikely	Low	**P3**	**L3**

2.3.5 Bibliography

[Dewhurst 2002]	Gotcha #5, “Misunderstanding References”
[ISO/IEC 14882-2014]	Subclause 8.3.2, “References”

2.4 DCL53-CPP. Do not write syntactically ambiguous declarations

It is possible to devise syntax that can ambiguously be interpreted as either an expression statement or a declaration. Syntax of this sort is called a *vexing parse* because the compiler must use disambiguation rules to determine the semantic results. The C++ Standard, [stmt.ambig], paragraph 1 [ISO/IEC 14882-2014], in part, states the following:

> There is an ambiguity in the grammar involving *expression-statement*s and declarations: An *expression-statement* with a function-style explicit type conversion as its leftmost subexpression can be indistinguishable from a declaration where the first declarator starts with a `(`. In those cases the statement is a declaration. [Note: To disambiguate, the whole statement might have to be examined to determine if it is an *expression-statement* or a declaration. ...

A similarly vexing parse exists within the context of a declaration where syntax can be ambiguously interpreted as either a function declaration or a declaration with a function-style cast as the initializer. The C++ Standard, [dcl.ambig.res], paragraph 1, in part, states the following:

> The ambiguity arising from the similarity between a function-style cast and a declaration mentioned in 6.8 can also occur in the context of a declaration. In that context, the choice is between a function declaration with a redundant set of parentheses around a parameter name and an object declaration with a function-style cast as the initializer. Just as for the ambiguities mentioned in 6.8, the resolution is to consider any construct that could possibly be a declaration a declaration.

Do not write a syntactically ambiguous declaration. With the advent of uniform initialization syntax using a *braced-init-list*, there is now syntax that unambiguously specifies a declaration instead of an expression statement. Declarations can also be disambiguated by using nonfunction-style casts, by initializing using =, or by removing extraneous parenthesis around the parameter name.

2.4.1 Noncompliant Code Example

In this noncompliant code example, an anonymous local variable of type `std::unique_lock` is expected to lock and unlock the mutex `m` by virtue of RAII. However, the declaration is syntactically ambiguous as it can be interpreted as declaring an anonymous object and calling its single-argument converting constructor or interpreted as declaring an object named `m` and default constructing it. The syntax used in this example defines the latter instead of the former, and so the mutex object is never locked.

```
#include <mutex>

static std::mutex m;
static int shared_resource;

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

2.4.2 Compliant Solution

In this compliant solution, the lock object is given an identifier (other than `m`) and the proper converting constructor is called.

```
#include <mutex>

static std::mutex m;
static int shared_resource;

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

2.4.3 Noncompliant Code Example

In this noncompliant code example, an attempt is made to declare a local variable, `w`, of type `Widget` while executing the default constructor. However, this declaration is syntactically ambiguous where the code could be either a declaration of a function pointer accepting no arguments and returning a `Widget` or a declaration of a local variable of type `Widget`. The syntax used in this example defines the former instead of the latter.

```
#include <iostream>

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

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

As a result, this program compiles and prints no output because the default constructor is never actually invoked.

2.4.4 Compliant Solution

This compliant solution shows two equally compliant ways to write the declaration. The first way is to elide the parentheses after the variable declaration, which ensures the syntax is that of a variable declaration instead of a function declaration. The second way is to use a *braced-init-list* to direct-initialize the local variable.

```
#include <iostream>

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

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

Running this program produces the output `Constructed` twice, once for `w1` and once for `w2`.

2.4.5 Noncompliant Code Example

This noncompliant code example demonstrates a vexing parse. The declaration `Gadget g(Widget(i));` is not parsed as declaring a `Gadget` object with a single argument. It is instead parsed as a function declaration with a redundant set of parentheses around a parameter.

```
#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;
}
```

Parentheses around parameter names are optional, so the following is a semantically identical spelling of the declaration.

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

As a result, this program is well-formed and prints only `3` as output because no `Gadget` or `Widget` objects are constructed.

2.4.6 Compliant Solution

This compliant solution demonstrates two equally compliant ways to write the declaration of `g`. The first declaration, `g1`, uses an extra set of parentheses around the argument to the constructor call, forcing the compiler to parse it as a local variable declaration of type `Gadget` instead of as a function declaration. The second declaration, `g2`, uses direct initialization to similar effect.

```
#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;
}
```

Running this program produces the expected output.

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

2.4.7 Risk Assessment

Syntactically ambiguous declarations can lead to unexpected program execution. However, it is likely that rudimentary testing would uncover violations of this rule.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
DCL53-CPP	Low	Unlikely	Medium	**P2**	**L3**

2.4.8 Bibliography

[ISO/IEC 14882-2014]	Subclause 6.8, “Ambiguity Resolution” Subclause 8.2, “Ambiguity Resolution”
[Meyers 2001]	Item 6, “Be Alert for C++’s Most Vexing Parse”

2.5 DCL54-CPP. Overload allocation and deallocation functions as a pair in the same scope

Allocation and deallocation functions can be overloaded at both global and class scopes.

If an allocation function is overloaded in a given scope, the corresponding deallocation function must also be overloaded in the same scope (and vice versa).

Failure to overload the corresponding dynamic storage function is likely to violate rules such as MEM51-CPP. Properly deallocate dynamically allocated resources. For instance, if an overloaded allocation function uses a private heap to perform its allocations, passing a pointer returned by it to the default deallocation function will likely cause undefined behavior. Even in situations in which the allocation function ultimately uses the default allocator to obtain a pointer to memory, failing to overload a corresponding deallocation function may leave the program in an unexpected state by not updating internal data for the custom allocator.

It is acceptable to define a deleted allocation or deallocation function without its corresponding free store function. For instance, it is a common practice to define a deleted non-placement allocation or deallocation function as a class member function when the class also defines a placement `new` function. This prevents accidental allocation via calls to `new` for that class type or deallocation via calls to `delete` on pointers to an object of that class type. It is acceptable to declare, but not define, a private allocation or deallocation function without its corresponding free store function for similar reasons. However, a definition must not be provided as that still allows access to the free store function within a class member function.

2.5.1 Noncompliant Code Example

In this noncompliant code example, an allocation function is overloaded at global scope. However, the corresponding deallocation function is not declared. Were an object to be allocated with the overloaded allocation function, any attempt to delete the object would result in undefined behavior in violation of MEM51-CPP. Properly deallocate dynamically allocated resources.

```
#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.
```

2.5.2 Compliant Solution

In this compliant solution, the corresponding deallocation function is also defined at global scope.

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

class HeapAllocator {
  static HANDLE h;
  static bool init;

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

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

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

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

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

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

2.5.3 Noncompliant Code Example

In this noncompliant code example, `operator new()` is overloaded at class scope, but `operator delete()` is not similarly overloaded at class scope. Despite that the overloaded allocation function calls through to the default global allocation function, were an object of type `S` to be allocated, any attempt to delete the object would result in leaving the program in an indeterminate state due to failing to update allocation bookkeeping accordingly.

```
#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;
  }
};
```

2.5.4 Compliant Solution

In this compliant solution, the corresponding `operator delete()` is overloaded at the same class scope.

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

2.5.5 Exceptions

DCL54-CPP-EX1: A placement deallocation function may be elided for a corresponding placement allocation function, but only if the object placement allocation and object construction are guaranteed to be `noexcept(true)`. Because placement deallocation functions are automatically invoked when the object initialization terminates by throwing an exception, it is safe to elide the placement deallocation function when exceptions cannot be thrown. For instance, some vendors implement compiler flags disabling exception support (such as -fno-cxx-exceptions in Clang and /EHs-c- in Microsoft Visual Studio), which has implementation-defined behavior when an exception is thrown but generally results in program termination similar to calling `abort()`.

2.5.6 Risk Assessment

Mismatched usage of `new` and `delete` could lead to a denial-of-service attack.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
DCL54-CPP	Low	Probable	Low	**P6**	**L2**

2.5.7 Related Guidelines

SEI CERT C++ Coding Standard	MEM51-CPP. Properly deallocate dynamically allocated resources

2.5.8 Bibliography

[ISO/IEC 14882-2014]	Subclause 3.7.4, “Dynamic Storage Duration” Subclause 5.3.4, “New” Subclause 5.3.5, “Delete”

2.6 DCL55-CPP. Avoid information leakage when passing a class object across a trust boundary

The C++ Standard, [class.mem], paragraph 13 [ISO/IEC 14882-2014], describes the layout of non-static data members of a non-union class, specifying the following:

> Nonstatic data members of a (non-union) class with the same access control are allocated so that later members have higher addresses within a class object. The order of allocation of non-static data members with different access control is unspecified. Implementation alignment requirements might cause two adjacent members not to be allocated immediately after each other; so might requirements for space for managing virtual functions and virtual base classes.

Further, [class.bit], paragraph 1, in part, states the following:

> Allocation of bit-fields within a class object is implementation-defined. Alignment of bit-fields is implementation-defined. Bit-fields are packed into some addressable allocation unit.

Thus, padding bits may be present at any location within a class object instance (including at the beginning of the object, in the case of an unnamed bit-field as the first member declared in a class). Unless initialized by zero-initialization, padding bits contain indeterminate values that may contain sensitive information.

When passing a pointer to a class object instance across a trust boundary to a different trusted domain, the programmer must ensure that the padding bits of such an object do not contain sensitive information.

2.6.1 Noncompliant Code Example

This noncompliant code example runs in kernel space and copies data from `arg` to user space. However, padding bits may be used within the object, for example, to ensure the proper alignment of class data members. These padding bits may contain sensitive information that may then be leaked when the data is copied to user space, regardless of how the data is copied.

```
#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));
}
```

2.6.2 Noncompliant Code Example

In this noncompliant code example, `arg` is value-initialized through direct initialization. Because `test` does not have a user-provided default constructor, the value-initialization is preceded by a zero-initialization that guarantees the padding bits are initialized to `0` before any further initialization occurs. It is akin to using `std::memset()` to initialize all of the bits in the object to `0`.

```
#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));
}
```

However, compilers are free to implement `arg.b = 2` by setting the low byte of a 32-bit register to `2`, leaving the high bytes unchanged, and storing all 32 bits of the register into memory. This could leak the high-order bytes resident in the register to a user.

2.6.3 Compliant Solution

This compliant solution serializes the structure data before copying it to an untrusted context.

```
#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 */);
}
```

This code ensures that no uninitialized padding bits are copied to unprivileged users. The structure copied to user space is now a packed structure and the `copy_to_user()` function would need to unpack it to recreate the original, padded structure.

2.6.4 Compliant Solution (Padding Bytes)

Padding bits can be explicitly declared as fields within the structure. This solution is not portable, however, because it depends on the implementation and target memory architecture. The following solution is specific to the x86-32 architecture.

```
#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));
}
```

The `static_assert()` declaration accepts a constant expression and an error message. The expression is evaluated at compile time and, if false, the compilation is terminated and the error message is used as the diagnostic. The explicit insertion of the padding bytes into the `struct` should ensure that no additional padding bytes are added by the compiler, and consequently both static assertions should be true. However, it is necessary to validate these assumptions to ensure that the solution is correct for a particular implementation.

2.6.5 Noncompliant Code Example

In this noncompliant code example, padding bits may abound, including

- alignment padding bits after a virtual method table or virtual base class data to align a subsequent data member,

- alignment padding bits to position a subsequent data member on a properly aligned boundary,
- alignment padding bits to position data members of varying access control levels.
- bit-field padding bits when the sequential set of bit-fields does not fill an entire allocation unit,
- bit-field padding bits when two adjacent bit-fields are declared with different underlying types,
- padding bits when a bit-field is declared with a length greater than the number of bits in the underlying allocation unit, or
- padding bits to ensure a class instance will be appropriately aligned for use within an array.

This code example runs in kernel space and copies data from `arg` to user space. However, the padding bits within the object instance may contain sensitive information that will then be leaked when the data is copied to user space.

```
#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));
}
```

Padding bits are implementation-defined, so the layout of the class object may differ between compilers or architectures. When compiled with GCC 5.3.0 for x86-32, the `test` object requires 96 bytes of storage to accommodate 29 bytes of data (33 bytes including the vtable) and has the following layout.

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

2.6.6 Compliant Solution

Due to the complexity of the data structure, this compliant solution serializes the object data before copying it to an untrusted context instead of attempting to account for all of the padding bytes manually.

```
#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;
    }

    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
  }
}
```

This code ensures that no uninitialized padding bits are copied to unprivileged users. The structure copied to user space is now a packed structure and the `copy_to_user()` function would need to unpack it to re-create the original, padded structure.

2.6.7 Risk Assessment

Padding bits might inadvertently contain sensitive data such as pointers to kernel data structures or passwords. A pointer to such a structure could be passed to other functions, causing information leakage.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
DCL55-CPP	Low	Unlikely	High	**P1**	**L3**

2.6.7.1 Related Vulnerabilities

Numerous vulnerabilities in the Linux Kernel have resulted from violations of this rule.

CVE-2010-4083 describes a vulnerability in which the `semctl()` system call allows unprivileged users to read uninitialized kernel stack memory because various fields of a `semid_ds struct` declared on the stack are not altered or zeroed before being copied back to the user.

CVE-2010-3881 describes a vulnerability in which structure padding and reserved fields in certain data structures in `QEMU-`KVM were not initialized properly before being copied to user space. A privileged host user with access to `/dev/kvm` could use this flaw to leak kernel stack memory to user space.

CVE-2010-3477 describes a kernel information leak in `act_police` where incorrectly initialized structures in the traffic-control dump code may allow the disclosure of kernel memory to user space applications.

2.6.8 Related Guidelines

SEI CERT C Coding Standard	DCL39-C. Avoid information leakage when passing a structure across a trust boundary

2.6.9 Bibliography

[ISO/IEC 14882-2014]	Subclause 8.5, “Initializers” Subclause 9.2, “Class Members” Subclause 9.6, “Bit-fields”

2.7 DCL56-CPP. Avoid cycles during initialization of static objects

The C++ Standard, [stmt.dcl], paragraph 4 [ISO/IEC 14882-2014], states the following:

> The zero-initialization (8.5) of all block-scope variables with static storage duration (3.7.1) or thread storage duration (3.7.2) is performed before any other initialization takes place. Constant initialization (3.6.2) of a block-scope entity with static storage duration, if applicable, is performed before its block is first entered. An implementation is permitted to perform early initialization of other block-scope variables with static or thread storage duration under the same conditions that an implementation is permitted to statically initialize a variable with static or thread storage duration in namespace scope (3.6.2). Otherwise such a variable is initialized the first time control passes through its declaration; such a variable is considered initialized upon the completion of its initialization. If the initialization exits by throwing an exception, the initialization is not complete, so it will be tried again the next time control enters the declaration. If control enters the declaration concurrently while the variable is being initialized, the concurrent execution shall wait for completion of the initialization. If control re-enters the declaration recursively while the variable is being initialized, the behavior is undefined.

Do not reenter a function during the initialization of a static variable declaration. If a function is reentered during the constant initialization of a static object inside that function, the behavior of the program is undefined. Infinite recursion is not required to trigger undefined behavior, the function need only recur once as part of the initialization. Due to thread-safe initialization of variables, a single, recursive call will often result in a deadlock due to locking a non-recursive synchronization primitive.

Additionally, the C++ Standard, [basic.start.init], paragraph 2, in part, states the following:

> Dynamic initialization of a non-local variable with static storage duration is either ordered or unordered. Definitions of explicitly specialized class template static data members have ordered initialization. Other class template static data members (i.e., implicitly or explicitly instantiated specializations) have unordered initialization. Other non-local variables with static storage duration have ordered initialization. Variables with ordered initialization defined within a single translation unit shall be initialized in the order of their definitions in the translation unit. If a program starts a thread, the subsequent initialization of a variable is unsequenced with respect to the initialization of a variable defined in a different translation unit. Otherwise, the initialization of a variable is indeterminately sequenced with respect to the initialization of a variable defined in a different translation unit. If a program starts a thread, the subsequent unordered initialization of a variable is unsequenced with respect to every other dynamic initialization. Otherwise, the unordered initialization of a variable is indeterminately sequenced with respect to every other dynamic initialization.

Do not create an initialization interdependency between static objects with dynamic initialization unless they are ordered with respect to one another. Unordered initialization, especially prevalent across translation unit boundaries, results in unspecified behavior.

2.7.1 Noncompliant Code Example

This noncompliant example attempts to implement an efficient factorial function using caching. Because the initialization of the static local array `cache` involves recursion, the behavior of the function is undefined, even though the recursion is not infinite.

```
#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;
}
```

2.7.1.1 Implementation Details

In Microsoft Visual Studio 2015 and GCC 6.1.0, the recursive initialization of `cache` deadlocks while initializing the static variable in a thread-safe manner.

2.7.2 Compliant Solution

This compliant solution avoids initializing the static local array `cache` and instead relies on zero-initialization to determine whether each member of the array has been assigned a value yet and, if not, recursively computes its value. It then returns the cached value when possible or computes the value as needed.

```
#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;
}
```

2.7.3 Noncompliant Code Example

In this noncompliant code example, the value of `numWheels` in `file1.cpp` relies on `c` being initialized. However, because `c` is defined in a different translation unit (`file2.cpp`) than `numWheels`, there is no guarantee that `c` will be initialized by calling `get_default_car()` before `numWheels` is initialized by calling `c.get_num_wheels()`. This is often referred to as the "static initialization order fiasco," and the resulting behavior is unspecified.

```
// 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();
```

2.7.3.1 Implementation Details

The value printed to the standard output stream will often rely on the order in which the translation units are linked. For instance, with Clang 3.8.0 on x86 Linux, the command `clang++ file1.cpp file2.cpp && ./a.out` will write `0` while `clang++ file2.cpp file1.cpp && ./a.out` will write `6`.

2.7.4 Compliant Solution

This compliant solution uses the “construct on first use” idiom to resolve the static initialization order issue. The code for `file.h` and `file2.cpp` are unchanged; only the static `numWheels` in `file1.cpp` is moved into the body of a function. Consequently, the initialization of `numWheels` is guaranteed to happen when control flows over the point of declaration, ensuring control over the order. The global object `c` is initialized before execution of `main()` begins, so by the time `get_num_wheels()` is called, `c` is guaranteed to have already been dynamically initialized.

```
// 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();
```

2.7.5 Risk Assessment

Recursively reentering a function during the initialization of one of its static objects can result in an attacker being able to cause a crash or denial of service. Indeterminately ordered dynamic initialization can lead to undefined behavior due to accessing an uninitialized object.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
DCL56-CPP	Low	Unlikely	Medium	**P2**	**L3**

2.7.6 Related Guideline

CERT Oracle Coding Standard for Java	DCL00-J. Prevent class initialization cycles

2.7.7 Bibliography

[ISO/IEC 14882-2014]	Subclause 3.6.2, “Initialization of Non-local Variables” Subclause 6.7, “Declaration Statement”

2.8 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.

2.8.1 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.

```
#include <stdexcept>

class S {
  bool has_error() const;

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

2.8.2 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.

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

2.8.3 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.

```
// 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.

```
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.

2.8.4 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.

```
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;
  }
};
```

2.8.5 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.

```
#include <stdexcept>

bool perform_dealloc(void *);

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

2.8.6 Compliant Solution

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

```
#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
  }
}
```

2.8.7 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.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
DCL57-CPP	Low	Likely	Medium	**P6**	**L3**

2.8.8 Related Guidelines

SEI CERT C++ Coding Standard	ERR55-CPP. Honor exception specifications ERR50-CPP. Do not abruptly terminate the program
MISRA C++:2008	Rule 15-5-1 (Required)

2.8.9 Bibliography

[Henricson 1997]	Recommendation 12.5, Do not let destructors called during stack unwinding throw exceptions
[ISO/IEC 14882-2014]	Subclause 3.4.7.2, “Deallocation Functions” 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)

2.9 DCL58-CPP. Do not modify the standard namespaces

Namespaces introduce new declarative regions for declarations, reducing the likelihood of conflicting identifiers with other declarative regions. One feature of namespaces is that they can be further extended, even within separate translation units. For instance, the following declarations are well-formed.

```
namespace MyNamespace {
int i;
}

namespace MyNamespace {
int i;
}

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

The standard library introduces the namespace `std` for standards-provided declarations such as `std::string`, `std::vector`, and `std::for_each`. However, it is undefined behavior to introduce new declarations in namespace `std` except under special circumstances. The C++ Standard, [namespace.std], paragraphs 1 and 2 [ISO/IEC 14882-2014], states the following:

> 1 The behavior of a C++ program is undefined if it adds declarations or definitions to namespace `std` or to a namespace within namespace `std` unless otherwise specified. A program may add a template specialization for any standard library template to namespace `std` only if the declaration depends on a user-defined type and the specialization meets the standard library requirements for the original template and is not explicitly prohibited.
>
> 2 The behavior of a C++ program is undefined if it declares
>
> - an explicit specialization of any member function of a standard library class template, or
> - an explicit specialization of any member function template of a standard library class or class template, or
> - an explicit or partial specialization of any member class template of a standard library class or class template.

In addition to restricting extensions to the namespace `std`, the C++ Standard, [namespace.posix], paragraph 1, further states the following:

> The behavior of a C++ program is undefined if it adds declarations or definitions to namespace `posix` or to a namespace within namespace `posix` unless otherwise specified. The namespace `posix` is reserved for use by ISO/IEC 9945 and other POSIX standards.

Do not add declarations or definitions to the standard namespaces `std` or `posix`, or to a namespace contained therein, except for a template specialization that depends on a user-defined type that meets the standard library requirements for the original template.

The Library Working Group, responsible for the wording of the Standard Library section of the C++ Standard, has an unresolved issue on the definition of *user-defined type*. Although the Library Working Group has no official stance on the definition [INCITS 2014], we define it to be any `class`, `struct`, `union`, or `enum` that is not defined within namespace `std` or a namespace contained within namespace `std`. Effectively, it is a user-provided type instead of a standard library–provided type.

2.9.1 Noncompliant Code Example

In this noncompliant code example, the declaration of `x` is added to the namespace `std`, resulting in undefined behavior.

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

2.9.2 Compliant Solution

This compliant solution assumes the intention of the programmer was to place the declaration of `x` into a namespace to prevent collisions with other global identifiers. Instead of placing the declaration into the namespace `std`, the declaration is placed into a namespace without a reserved name.

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

2.9.3 Noncompliant Code Example

In this noncompliant code example, a template specialization of `std::plus` is added to the namespace `std` in an attempt to allow `std::plus` to concatenate a `std::string` and `MyString` object. However, because the template specialization is of a standard library–provided type (`std::string`), this code results in undefined behavior.

```
#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();
  }
};
}

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

  std::cout << p(s1, s2) << std::endl;
}
```

2.9.4 Compliant Solution

The interface for `std::plus` requires that both arguments to the function call operator and the return type are of the same type. Because the attempted specialization in the noncompliant code example results in undefined behavior, this compliant solution defines a new `std::binary_function` derivative that can add a `std::string` to a `MyString` object without requiring modification of the namespace `std`.

```
#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;
}
```

2.9.5 Compliant Solution

In this compliant solution, a specialization of `std::plus` is added to the `std` namespace, but the specialization depends on a user-defined type and meets the Standard Template Library requirements for the original template, so it complies with this rule. However, because `MyString` can be constructed from `std::string`, this compliant solution involves invoking a converting constructor whereas the previous compliant solution does not.

```
#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;
}
```

2.9.6 Risk Assessment

Altering the standard namespace can cause undefined behavior in the C++ standard library.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
DCL58-CPP	High	Unlikely	Medium	**P6**	**L2**

2.9.7 Related Guidelines

SEI CERT C++ Coding Standard	DCL51-CPP. Do not declare or define a reserved identifier

2.9.8 Bibliography

[INCITS 2014]	Issue 2139, "What Is a *User-Defined* Type?"
[ISO/IEC 14882-2014]	Subclause 17.6.4.2.1, "Namespace `std`" Subclause 17.6.4.2.2, "Namespace `posix`"

2.10 DCL59-CPP. Do not define an unnamed namespace in a header file

Unnamed namespaces are used to define a namespace that is unique to the translation unit, where the names contained within have internal linkage by default. The C++ Standard, [namespace.unnamed], paragraph 1 [ISO/IEC 14882-2014], states the following:

> An *unnamed-namespace-definition* behaves as if it were replaced by:
>
> inline namespace unique { /* empty body */ }
> using namespace unique ;
> namespace unique { namespace-body }
>
> where `inline` appears if and only if it appears in the *unnamed-namespace-definition*, all occurrences of `unique` in a translation unit are replaced by the same identifier, and this identifier differs from all other identifiers in the entire program.

Production-quality C++ code frequently uses *header files* as a means to share code between translation units. A header file is any file that is inserted into a translation unit through an `#include` directive. Do not define an unnamed namespace in a header file. When an unnamed namespace is defined in a header file, it can lead to surprising results. Due to default internal linkage, each translation unit will define its own unique instance of members of the unnamed namespace that are ODR-used within that translation unit. This can cause unexpected results, bloat the resulting executable, or inadvertently trigger undefined behavior due to one-definition rule (ODR) violations.

2.10.1 Noncompliant Code Example

In this noncompliant code example, the variable `v` is defined in an unnamed namespace within a header file and is accessed from two separate translation units. Each translation unit prints the current value of `v` and then assigns a new value into it. However, because `v` is defined within an unnamed namespace, each translation unit operates on its own instance of `v`, resulting in unexpected output.

```
// a.h
#ifndef A_HEADER_FILE
#define A_HEADER_FILE

namespace {
int v;
}

#endif // A_HEADER_FILE

// a.cpp
#include "a.h"
#include <iostream>

void f() {
  std::cout << "f(): " << v << std::endl;
  v = 42;
  // ...
}

// b.cpp
#include "a.h"
#include <iostream>

void g() {
  std::cout << "g(): " << v << std::endl;
  v = 100;
}

int main() {
  extern void f();
  f(); // Prints v, sets it to 42
  g(); // Prints v, sets it to 100
  f();
  g();
}
```

When executed, this program prints the following.

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

2.10.2 Compliant Solution

In this compliant solution, `v` is defined in only one translation unit but is externally visible to all translation units, resulting in the expected behavior.

```
// a.h
#ifndef A_HEADER_FILE
#define A_HEADER_FILE

extern int v;

#endif // A_HEADER_FILE

// a.cpp
#include "a.h"
#include <iostream>

int v; // Definition of global variable v

void f() {
  std::cout << "f(): " << v << std::endl;
  v = 42;
  // ...
}

// b.cpp
#include "a.h"
#include <iostream>

void g() {
  std::cout << "g(): " << v << std::endl;
  v = 100;
}

int main() {
  extern void f();
  f(); // Prints v, sets it to 42
  g(); // Prints v, sets it to 100
  f(); // Prints v, sets it back to 42
  g(); // Prints v, sets it back to 100
}
```

When executed, this program prints the following.

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

2.10.3 Noncompliant Code Example

In this noncompliant code example, the variable `v` is defined in an unnamed namespace within a header file, and an inline function, `get_v()`, is defined, which accesses that variable. ODR-using the inline function from multiple translation units (as shown in the implementation of `f()` and `g()`) violates the one-definition rule because the definition of `get_v()` is not identical in all translation units due to referencing a unique `v` in each translation unit.

```
// 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"

void f() {
  int i = get_v();
  // ...
}

// b.cpp
#include "a.h"

void g() {
  int i = get_v();
  // ...
}
```

See DCL60-CPP. Obey the one-definition rule for more information on violations of the one-definition rule.

2.10.4 Compliant Solution

In this compliant solution, `v` is defined in only one translation unit but is externally visible to all translation units and can be accessed from the inline `get_v()` function.

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

2.10.5 Noncompliant Code Example

In this noncompliant code example, the function `f()` is defined within a header file. However, including the header file in multiple translation units causes a violation of the one-definition rule that usually results in an error diagnostic generated at link time due to multiple definitions of a function with the same name.

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

2.10.6 Noncompliant Code Example

This noncompliant code example attempts to resolve the link-time errors by defining `f()` within an unnamed namespace. However, it produces multiple, unique definitions of `f()` in the resulting executable. If `a.h` is included from many translation units, it can lead to increased link times, a larger executable file, and reduced performance.

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

2.10.7 Compliant Solution

In this compliant solution, `f()` is not defined with an unnamed namespace and is instead defined as an inline function. Inline functions are required to be defined identically in all the translation units in which they are used, which allows an implementation to generate only a single instance of the function at runtime in the event the body of the function does not get generated for each call site.

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

2.10.8 Risk Assessment

Defining an unnamed namespace within a header file can cause data integrity violations and performance problems but is unlikely to go unnoticed with sufficient testing. One-definition rule violations result in undefined behavior.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
DCL59-CPP	Medium	Unlikely	Medium	**P4**	**L3**

2.10.9 Related Guidelines

SEI CERT C++ Coding Standard	DCL60-CPP. Obey the one-definition rule

2.10.10 Bibliography

[ISO/IEC 14882-2014]	Subclause 3.2, “One Definition Rule” Subclause 7.1.2, “Function Specifiers” Subclause 7.3.1, “Namespace Definition”

2.11 DCL60-CPP. Obey the one-definition rule

Nontrivial C++ programs are generally divided into multiple translation units that are later linked together to form an executable. To support such a model, C++ restricts named object definitions to ensure that linking will behave deterministically by requiring a single definition for an object across all translation units. This model is called the o*ne-definition rule* (ODR), which is defined by the C++ Standard, [basic.def.odr] in paragraph 4 [ISO/IEC 14882-2014]:

> Every program shall contain exactly one definition of every non-inline function or variable that is odr-used in that program; no diagnostic required. The definition can appear explicitly in the program, it can be found in the standard or a user-defined library, or (when appropriate) it is implicitly defined. An inline function shall be defined in every translation unit in which it is odr-used.

The most common approach to multitranslation unit compilation involves declarations residing in a header file that is subsequently made available to a source file via `#include`. These declarations are often also definitions, such as class and function template definitions. This approach is allowed by an exception defined in paragraph 6, which, in part, states the following:

> There can be more than one definition of a class type, enumeration type, inline function with external linkage, class template, non-static function template, static data member of a class template, member function of a class template, or template specialization for which some template parameters are not specified in a program provided that each definition appears in a different translation unit, and provided the definitions satisfy the following requirements. Given such an entity named `D` defined in more than one translation unit....
>
> If the definitions of `D` satisfy all these requirements, then the program shall behave as if there were a single definition of `D`. If the definitions of `D` do not satisfy these requirements, then the behavior is undefined.

The requirements specified by paragraph 6 essentially state that that two definitions must be identical (not simply equivalent). Consequently, a definition introduced in two separate translation units by an `#include` directive generally will not violate the ODR because the definitions are identical in both translation units.

However, it is possible to violate the ODR of a definition introduced via `#include` using block language linkage specifications, vendor-specific language extensions, and so on. A more likely scenario for ODR violations is that accidental definitions of differing objects will exist in different translation units.

Do not violate the one-definition rule; violations result in undefined behavior.

2.11.1 Noncompliant Code Example

In this noncompliant code example, two different translation units define a class of the same name with differing definitions. Although the two definitions are functionally equivalent (they both define a class named `S` with a single, public, nonstatic data member `int a`), they are not defined using the same sequence of tokens. This code example violates the ODR and results in undefined behavior.

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

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

2.11.2 Compliant Solution

The correct mitigation depends on programmer intent. If the programmer intends for the same class definition to be visible in both translation units because of common usage, the solution is to use a header file to introduce the object into both translation units, as shown in this compliant solution.

```
// S.h
struct S {
  int a;
};

// a.cpp
#include "S.h"

// b.cpp
#include "S.h"
```

2.11.3 Compliant Solution

If the ODR violation was a result of accidental name collision, the best mitigation solution is to ensure that both class definitions are unique, as in this compliant solution.

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

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

Alternatively, the classes could be given distinct names in each translation unit to avoid violating the ODR.

2.11.4 Noncompliant Code Example (Microsoft Visual Studio)

In this noncompliant code example, a class definition is introduced into two translation units using `#include`. However, one of the translation units uses an implementation-defined `#pragma` that is supported by Microsoft Visual Studio to specify structure field alignment requirements. Consequently, the two class definitions may have differing layouts in each translation unit, which is a violation of the ODR.

```
// 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);
}
```

2.11.4.1 Implementation Details

It is possible for the preceding noncompliant code example to result in `a.cpp` allocating space for an object with a different size than expected by `init_s()` in `s.cpp`. When translating `s.cpp`, the layout of the structure may include padding bytes between the `c` and `a` data members. When translating `a.cpp`, the layout of the structure may remove those padding bytes as a result of the `#pragma pack` directive, so the object passed to `init_s()` may be smaller than expected. Consequently, when `init_s()` initializes the data members of `s`, it may result in a buffer overrun.

For more information on the behavior of `#pragma pack`, see the vendor documentation for your implementation, such as Microsoft Visual Studio or GCC.

2.11.5 Compliant Solution

In this compliant solution, the implementation-defined structure member-alignment directive is removed, ensuring that all definitions of `S` comply with the ODR.

```
// 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
#include "s.h"

void f() {
  S s;
  init_s(s);
}
```

2.11.6 Noncompliant Code Example

In this noncompliant code example, the constant object `n` has internal linkage but is odr-used within `f()`, which has external linkage. Because `f()` is declared as an inline function, the definition of `f()` must be identical in all translation units. However, each translation unit has a unique instance of `n`, resulting in a violation of the ODR.

```
const int n = 42;

int g(const int &lhs, const int &rhs);

inline int f(int k) {
  return g(k, n);
}
```

2.11.7 Compliant Solution

A compliant solution must change one of three factors: (1) it must not odr-use `n` within `f()`, (2) it must declare `n` such that it has external linkage, or (3) it must not use an inline definition of `f()`.

If circumstances allow modification of the signature of `g()` to accept parameters by value instead of by reference, then `n` will not be odr-used within `f()` because `n` would then qualify as a constant expression. This solution is compliant but it is not ideal. It may not be possible (or desirable) to modify the signature of `g()`, such as if `g()` represented `std::max()` from `<algorithm>`. Also, because of the differing linkage used by `n` and `f()`, accidental violations of the ODR are still likely if the definition of `f()` is modified to odr-use `n`.

```
const int n = 42;

int g(int lhs, int rhs);

inline int f(int k) {
  return g(k, n);
}
```

2.11.8 Compliant Solution

In this compliant solution, the constant object `n` is replaced with an enumerator of the same name. Named enumerations defined at namespace scope have the same linkage as the namespace they are contained in. The global namespace has external linkage, so the definition of the named enumeration and its contained enumerators also have external linkage. Although less aesthetically pleasing, this compliant solution does not suffer from the same maintenance burdens of the previous code because `n` and `f()` have the same linkage.

```
enum Constants {
  N = 42
};

int g(const int &lhs, const int &rhs);

inline int f(int k) {
  return g(k, N);
}
```

2.11.9 Risk Assessment

Violating the ODR causes undefined behavior, which can result in exploits as well as denial-of-service attacks. As shown in “Support for Whole-Program Analysis and the Verification of the One-Definition Rule in C++” [Quinlan 2006], failing to enforce the ODR enables a virtual function pointer attack known as the *VPTR* exploit. In this exploit, an object’s virtual function table is corrupted so that calling a virtual function on the object results in malicious code being executed. See the paper by Quinlan and colleagues for more details. However, note that to introduce the malicious class, the attacker must have access to the system building the code.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
DCL60-CPP	High	Unlikely	High	**P3**	**L3**

2.11.10 Bibliography

[ISO/IEC 14882-2014]	Subclause 3.2, “One Definition Rule”
[Quinlan 2006]	

3 Expressions (EXP)

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

In C++, modifying an object, calling a library I/O function, accessing a `volatile`-qualified value, or calling a function that performs one of these actions are ways to modify the state of the execution environment. These actions are called *side effects*. All relationships between value computations and side effects can be described in terms of sequencing of their evaluations. The C++ Standard, [intro.execution], paragraph 13 [ISO/IEC 14882-2014], establishes three sequencing terms:

> *Sequenced before* is an asymmetric, transitive, pair-wise relation between evaluations executed by a single thread, which induces a partial order among those evaluations. Given any two evaluations *A* and *B*, if *A* is sequenced before *B*, then the execution of *A* shall precede the execution of *B*. If *A* is not sequenced before *B* and *B* is not sequenced before *A*, then *A* and *B* are *unsequenced*. [Note: The execution of unsequenced evaluations can overlap. — end note] Evaluations *A* and *B* are *indeterminately sequenced* when either *A* is sequenced before *B* or *B* is sequenced before *A*, but it is unspecified which. [Note: Indeterminately sequenced evaluations cannot overlap, but either could be executed first. — end note]

Paragraph 15 further states (nonnormative text removed for brevity) the following:

> Except where noted, evaluations of operands of individual operators and of subexpressions of individual expressions are unsequenced. ... The value computations of the operands of an operator are sequenced before the value computation of the result of the operator. If a side effect on a scalar object is unsequenced relative to either another side effect on the same scalar object or a value computation using the value of the same scalar object, and they are not potentially concurrent, the behavior is undefined. ... When calling a function (whether or not the function is inline), every value computation and side effect associated with any argument expression, or with the postfix expression designating the called function, is sequenced before execution of every expression or statement in the body of the called function. ... Every evaluation in the calling function (including other function calls) that is not otherwise specifically sequenced before or after the execution of the body of the called function is indeterminately sequenced with respect to the execution of the called function. Several contexts in C++ cause evaluation of a function call, even though no corresponding function call syntax appears in the translation unit. ... The sequencing constraints on the execution of the called function (as described above) are features of the function calls as evaluated, whatever the syntax of the expression that calls the function might be.

Do not allow the same scalar object to appear in side effects or value computations in both halves of an unsequenced or indeterminately sequenced operation.

The following expressions have sequencing restrictions that deviate from the usual unsequenced ordering [ISO/IEC 14882-2014]:

- In postfix `++` and `--` expressions, the value computation is sequenced before the modification of the operand. ([expr.post.incr], paragraph 1)
- In logical && expressions, if the second expression is evaluated, every value computation and side effect associated with the first expression is sequenced before every value computation and side effect associated with the second expression. ([expr.log.and], paragraph 2)
- In logical `||` expressions, if the second expression is evaluated, every value computation and side effect associated with the first expression is sequenced before every value computation and side effect associated with the second expression. ([expr.log.or], paragraph 2)
- In conditional `?:` expressions, every value computation and side effect associated with the first expression is sequenced before every value computation and side effect associated with the second or third expression (whichever is evaluated). ([expr.cond], paragraph 1)
- In assignment expressions (including compound assignments), the assignment is sequenced after the value computations of left and right operands and before the value computation of the assignment expression. ([expr.ass], paragraph 1)
- In comma `,` expressions, every value computation and side effect associated with the left expression is sequenced before every value computation and side effect associated with the right expression. ([expr.comma], paragraph 1)
- When evaluating initializer lists, the value computation and side effect associated with each *initializer-clause* is sequenced before every value computation and side effect associated with a subsequent *initializer-clause*. ([dcl.init.list], paragraph 4)
- When a signal handler is executed as a result of a call to `std::raise()`, the execution of the handler is sequenced after the invocation of `std::raise()` and before its return. ([intro.execution], paragraph 6)
- The completions of the destructors for all initialized objects with thread storage duration within a thread are sequenced before the initiation of the destructors of any object with static storage duration. ([basic.start.term], paragraph 1)
- In a *new-expression*, initialization of an allocated object is sequenced before the value computation of the *new-expression*. ([expr.new], paragraph 18)
- When a default constructor is called to initialize an element of an array and the constructor has at least one default argument, the destruction of every temporary created in a default argument is sequenced before the construction of the next array element, if any. ([class.temporary], paragraph 4)
- The destruction of a temporary whose lifetime is not extended by being bound to a reference is sequenced before the destruction of every temporary that is constructed earlier in the same full-expression. ([class.temporary], paragraph 5)
- Atomic memory ordering functions can explicitly determine the sequencing order for expressions. ([atomics.order] and [atomics.fences])

This rule means that statements such as

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

have defined behavior, and statements such as the following do not.

```
// 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;
```

Not all instances of a comma in C++ code denote use of the comma operator. For example, the comma between arguments in a function call is *not* the comma operator. Additionally, overloaded operators behave the same as a function call, with the operands to the operator acting as arguments to a function call.

3.1.1 Noncompliant Code Example

In this noncompliant code example, `i` is evaluated more than once in an unsequenced manner, so the behavior of the expression is undefined.

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

3.1.2 Compliant Solution

These examples are independent of the order of evaluation of the operands and can each be interpreted in only one way.

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

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

3.1.3 Noncompliant Code Example

The call to `func()` in this noncompliant code example has undefined behavior because the argument expressions are unsequenced.

```
extern void func(int i, int j);

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

The first (left) argument expression reads the value of `i` (to determine the value to be stored) and then modifies `i`. The second (right) argument expression reads the value of `i`, but not to determine the value to be stored in `i`. This additional attempt to read the value of `i` has undefined behavior.

3.1.4 Compliant Solution

This compliant solution is appropriate when the programmer intends for both arguments to `func()` to be equivalent.

```
extern void func(int i, int j);

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

This compliant solution is appropriate when the programmer intends for the second argument to be 1 greater than the first.

```
extern void func(int i, int j);

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

3.1.5 Noncompliant Code Example

This noncompliant code example is similar to the previous noncompliant code example. However, instead of calling a function directly, this code calls an overloaded `operator<<()`. Overloaded operators are equivalent to a function call and have the same restrictions regarding the sequencing of the function call arguments. This means that the operands are not evaluated left-to-right, but are unsequenced with respect to one another. Consequently, this noncompliant code example has undefined behavior.

```
#include <iostream>

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

3.1.6 Compliant Solution

In this compliant solution, two calls are made to `operator<<()`, ensuring that the arguments are printed in a well-defined order.

```
#include <iostream>

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

3.1.7 Noncompliant Code Example

The order of evaluation for function arguments is unspecified. This noncompliant code example exhibits unspecified behavior but not undefined behavior.

```
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());
}
```

The order in which `a()` and `b()` are called is unspecified; the only guarantee is that both `a()` and `b()` will be called before `c()` is called. If `a()` or `b()` rely on shared state when calculating their return value, as they do in this example, the resulting arguments passed to `c()` may differ between compilers or architectures.

3.1.8 Compliant Solution

In this compliant solution, the order of evaluation for `a()` and `b()` is fixed, and so no unspecified behavior occurs.

```
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);
}
```

3.1.9 Risk Assessment

Attempting to modify an object in an unsequenced or indeterminately sequenced evaluation may cause that object to take on an unexpected value, which can lead to unexpected program behavior.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
EXP50-CPP	Medium	Probable	Medium	**P8**	**L2**

3.1.10 Related Guidelines

SEI CERT C Coding Standard	EXP30-C. Do not depend on the order of evaluation for side effects

3.1.11 Bibliography

[ISO/IEC 14882-2014]	Subclause 1.9, “Program Execution”
[MISRA 2008]	Rule 5-0-1 (Required)

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

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

> In the first alternative (*delete object*), if the static type of the object to be deleted is different from its dynamic type, the static type shall be a base class of the dynamic type of the object to be deleted and the static type shall have a virtual destructor or the behavior is undefined. In the second alternative (*delete array*) if the dynamic type of the object to be deleted differs from its static type, the behavior is undefined.

Do not delete an array object through a static pointer type that differs from the dynamic pointer type of the object. Deleting an array through a pointer to the incorrect type results in undefined behavior.

3.2.1 Noncompliant Code Example

In this noncompliant code example, an array of `Derived` objects is created and the pointer is stored in a `Base *`. Despite `Base::~Base`() being declared virtual, it still results in undefined behavior. Further, attempting to perform pointer arithmetic on the static type `Base *` violates CTR56-CPP. Do not use pointer arithmetic on polymorphic objects.

```
struct Base {
  virtual ~Base() = default;
};

struct Derived final : Base {};

void f() {
   Base *b = new Derived[10];
   // ...
   delete [] b;
}
```

3.2.2 Compliant Solution

In this compliant solution, the static type of `b` is `Derived *`, which removes the undefined behavior when indexing into the array as well as when deleting the pointer.

```
struct Base {
  virtual ~Base() = default;
};

struct Derived final : Base {};

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

3.2.3 Risk Assessment

Attempting to destroy an array of polymorphic objects through the incorrect static type is undefined behavior. In practice, potential consequences include abnormal program execution and memory leaks.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
EXP51-CPP	Low	Unlikely	Medium	**P2**	**L3**

3.2.4 Related Guidelines

SEI CERT C++ Coding Standard	CTR56-CPP. Do not use pointer arithmetic on polymorphic objects OOP52-CPP. Do not delete a polymorphic object without a virtual destructor

3.2.5 Bibliography

[ISO/IEC 14882-2014]	Subclause 5.3.5, “Delete”

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

Some expressions involve operands that are *unevaluated*. The C++ Standard, [expr], paragraph 8 [ISO/IEC 14882-2014] states the following:

> In some contexts, *unevaluated operands* appear. An unevaluated operand is not evaluated. An unevaluated operand is considered a full-expression. [Note: In an unevaluated operand, a non-static class member may be named (5.1) and naming of objects or functions does not, by itself, require that a definition be provided. — end note]

The following expressions do not evaluate their operands: `sizeof()`, `typeid()`, `noexcept()`, `decltype()`, and `declval()`.

Because an unevaluated operand in an expression is not evaluated, no side effects from that operand are triggered. Reliance on those side effects will result in unexpected behavior. Do not rely on side effects in unevaluated operands.

Unevaluated expression operands are used when the declaration of an object is required but the definition of the object is not. For instance, in the following example, the function `f()` is overloaded, relying on the unevaluated expression operand to select the desired overload, which is then used to determine the result of the `sizeof()` expression.

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

Such a use does not rely on the side effects of `f()` and consequently conforms to this guideline.

3.3.1 Noncompliant Code Example (`sizeof`)

In this noncompliant code example, the expression `a++` is not evaluated.

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

Consequently, the value of `a` after `b` has been initialized is 14.

3.3.2 Compliant Solution (`sizeof`)

In this compliant solution, the variable `a` is incremented outside of the `sizeof` operator.

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

3.3.3 Noncompliant Code Example (`decltype`)

In this noncompliant code example, the expression `i++` is not evaluated within the `decltype` specifier.

```
#include <iostream>

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

Consequently, the value of i remains 0.

3.3.4 Compliant Solution (`decltype`)

In this compliant solution, `i` is incremented outside of the `decltype` specifier so that it is evaluated as desired.

```
#include <iostream>

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

3.3.5 Exceptions

EXP52-CPP-EX1: It is permissible for an expression with side effects to be used as an unevaluated operand in a macro definition or SFINAE context. Although these situations rely on the side effects to produce valid code, they typically do not rely on values produced as a result of the side effects.

The following code is an example of compliant code using an unevaluated operand in a macro definition.

```
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);
}
```

The expansion of the macro `m` will result in the expression `++i` being used as an unevaluated operand to `sizeof()`. However, the expectation of the programmer at the expansion loci is that `i` is preincremented only once. Consequently, this is a safe macro and complies with PRE31-C. Avoid side effects in arguments to unsafe macros. Compliance with that rule is especially important for code that follows this exception.

The following code is an example of compliant code using an unevaluated operand in a SFINAE context to determine whether a type can be postfix incremented.

```
#include <iostream>
#include <type_traits>
#include <utility>

template <typename T>
class is_incrementable {
  typedef char one[1];
  typedef char two[2];
  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;
}
```

In an instantiation of `is_incrementable`, the use of the postfix increment operator generates side effects that are used to determine whether the type is postfix incrementable. However, the value result of these side effects is discarded, so the side effects are used only for SFINAE.

3.3.6 Risk Assessment

If expressions that appear to produce side effects are an unevaluated operand, the results may be different than expected. Depending on how this result is used, it can lead to unintended program behavior.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
EXP52-CPP	Low	Unlikely	Low	**P3**	**L3**

3.3.7 Related Guidelines

SEI CERT C Coding Standard	EXP44-C. Do not rely on side effects in operands to sizeof, _Alignof, or _Generic

3.3.8 Bibliography

[ISO/IEC 14882-2014]	Clause 5, "Expressions" Subclause 20.2.5, "Function Template `declval`"

3.4 EXP53-CPP. Do not read uninitialized memory

Local, automatic variables assume unexpected values if they are read before they are initialized. The C++ Standard, [dcl.init], paragraph 12 [ISO/IEC 14882-2014], states the following:

> If no initializer is specified for an object, the object is default-initialized. When storage for an object with automatic or dynamic storage duration is obtained, the object has an *indeterminate value*, and if no initialization is performed for the object, that object retains an indeterminate value until that value is replaced. If an indeterminate value is produced by an evaluation, the behavior is undefined except in the following cases:
>
> - If an indeterminate value of unsigned narrow character type is produced by the evaluation of:
>   — the second or third operand of a conditional expression,
>   — the right operand of a comma expression,
>   — the operand of a cast or conversion to an unsigned narrow character type, or
>   — a discarded-value expression,
>   then the result of the operation is an indeterminate value.
>
> - If an indeterminate value of unsigned narrow character type is produced by the evaluation of the right operand of a simple assignment operator whose first operand is an lvalue of unsigned narrow character type, an indeterminate value replaces the value of the object referred to by the left operand.
>
> - If an indeterminate value of unsigned narrow character type is produced by the evaluation of the initialization expression when initializing an object of unsigned narrow character type, that object is initialized to an indeterminate value.

The default initialization of an object is described by paragraph 7 of the same subclause:

> To *default-initialize* an object of type T means:
>
> - if T is a (possibly cv-qualified) class type, the default constructor for T is called (and the initialization is ill-formed if T has no default constructor or overload resolution results in an ambiguity or in a function that is deleted or inaccessible from the context of the initialization);
>
> - if T is an array type, each element is default-initialized;
>
> - otherwise, no initialization is performed.
>
> If a program calls for the default initialization of an object of a const-qualified type T, T shall be a class type with a user-provided default constructor.

As a result, objects of type `T` with automatic or dynamic storage duration must be explicitly initialized before having their value read as part of an expression unless `T` is a class type or an array thereof or is an unsigned narrow character type. If `T` is an unsigned narrow character type, it may be used to initialize an object of unsigned narrow character type, which results in both objects having an indeterminate value. This technique can be used to implement copy operations such as `std::memcpy()` without triggering undefined behavior.

Additionally, memory dynamically allocated with a `new` expression is default-initialized when the *new-initialized* is omitted. Memory allocated by the standard library function `std::calloc()` is zero-initialized. Memory allocated by the standard library function `std::realloc()` assumes the values of the original pointer but may not initialize the full range of memory. Memory allocated by any other means (`std::malloc()`, allocator objects, `operator new()`, and so on) is assumed to be default-initialized.

Objects of static or thread storage duration are zero-initialized before any other initialization takes place [ISO/IEC 14882-2014] and need not be explicitly initialized before having their value read.

Reading uninitialized variables for creating entropy is problematic because these memory accesses can be removed by compiler optimization. VU#925211 is an example of a vulnerability caused by this coding error [VU#925211].

3.4.1 Noncompliant Code Example

In this noncompliant code example, an uninitialized local variable is evaluated as part of an expression to print its value, resulting in undefined behavior.

```
#include <iostream>

void f() {
  int i;
  std::cout << i;
}
```

3.4.2 Compliant Solution

In this compliant solution, the object is initialized prior to printing its value.

```
#include <iostream>

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

3.4.3 Noncompliant Code Example

In this noncompliant code example, an `int *` object is allocated by a *new-expression*, but the memory it points to is not initialized. The object's pointer value and the value it points to are printed to the standard output stream. Printing the pointer value is well-defined, but attempting to print the value pointed to yields an indeterminate value, resulting in undefined behavior.

```
#include <iostream>

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

3.4.4 Compliant Solution

In this compliant solution, the memory is direct-initialized to the value `12` prior to printing its value.

```
#include <iostream>

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

Initialization of an object produced by a *new-expression* is performed by placing (possibly empty) parenthesis or curly braces after the type being allocated. This causes direct initialization of the pointed-to object to occur, which will zero-initialize the object if the initialization omits a value, as illustrated by the following code.

```
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
```

3.4.5 Noncompliant Code Example

In this noncompliant code example, the class member variable `c` is not explicitly initialized by a *ctor-initializer* in the default constructor. Despite the local variable `s` being default-initialized, the use of `c` within the call to `S::f()` results in the evaluation of an object with indeterminate value, resulting in undefined behavior.

```
class S {
  int c;

public:
  int f(int i) const { return i + c; }
};

void f() {
  S s;
  int i = s.f(10);
}
```

3.4.6 Compliant Solution

In this compliant solution, `S` is given a default constructor that initializes the class member variable `c`.

```
class S {
  int c;

public:
  S() : c(0) {}
  int f(int i) const { return i + c; }
};

void f() {
  S s;
  int i = s.f(10);
}
```

3.4.7 Risk Assessment

Reading uninitialized variables is undefined behavior and can result in unexpected program behavior. In some cases, these security flaws may allow the execution of arbitrary code.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
EXP53-CPP	High	Probable	Medium	**P12**	**L1**

3.4.8 Related Guidelines

SEI CERT C Coding Standard	EXP33-C. Do not read uninitialized memory

3.4.9 Bibliography

[ISO/IEC 14882-2014]	Clause 5, “Expressions” Subclause 5.3.4, “New” Subclause 8.5, “Initializers” Subclause 12.6.2, “Initializing Bases and Members”
[Lockheed Martin 2005]	Rule 142, All variables shall be initialized before use

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

Every object has a lifetime in which it can be used in a well-defined manner. The lifetime of an object begins when sufficient, properly aligned storage has been obtained for it and its initialization is complete. The lifetime of an object ends when a nontrivial destructor, if any, is called for the object and the storage for the object has been reused or released. Use of an object, or a pointer to an object, outside of its lifetime frequently results in undefined behavior.

The C++ Standard, [basic.life], paragraph 5 [ISO/IEC 14882-2014], describes the lifetime rules for pointers:

> Before the lifetime of an object has started but after the storage which the object will occupy has been allocated or, after the lifetime of an object has ended and before the storage which the object occupied is reused or released, any pointer that refers to the storage location where the object will be or was located may be used but only in limited ways. For an object under construction or destruction, see 12.7. Otherwise, such a pointer refers to allocated storage, and using the pointer as if the pointer were of type `void*`, is well-defined. Indirection through such a pointer is permitted but the resulting lvalue may only be used in limited ways, as described below. The program has undefined behavior if:
>
> - the object will be or was of a class type with a non-trivial destructor and the pointer is used as the operand of a *delete-expression*,
> - the pointer is used to access a non-static data member or call a non-static member function of the object, or
> - the pointer is implicitly converted to a pointer to a virtual base class, or
> - the pointer is used as the operand of a `static_cast`, except when the conversion is to pointer to *cv* `void`, or to pointer to *cv* `void` and subsequently to pointer to either *cv* `char` or *cv* `unsigned char`, or
> - the pointer is used as the operand of a `dynamic_cast`.

Paragraph 6 describes the lifetime rules for non-pointers:

> Similarly, before the lifetime of an object has started but after the storage which the object will occupy has been allocated or, after the lifetime of an object has ended and before the storage which the object occupied is reused or released, any glvalue that refers to the original object may be used but only in limited ways. For an object under construction or destruction, see 12.7. Otherwise, such a glvalue refers to allocated storage, and using the properties of the glvalue that do not depend on its value is well-defined.
>
> The program has undefined behavior if:
>
> - an lvalue-to-rvalue conversion is applied to such a glvalue,
> - the glvalue is used to access a non-static data member or call a non-static member function of the object, or
> - the glvalue is bound to a reference to a virtual base class, or
> - the glvalue is used as the operand of a `dynamic_cast` or as the operand of `typeid`.

Do not use an object outside of its lifetime, except in the ways described above as being well-defined.

3.5.1 Noncompliant Code Example

In this noncompliant code example, a pointer to an object is used to call a non-static member function of the object prior to the beginning of the pointer's lifetime, resulting in undefined behavior.

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

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

3.5.2 Compliant Solution

In this compliant solution, storage is obtained for the pointer prior to calling `S::mem_fn()`.

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

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

An improved compliant solution would not dynamically allocate memory directly but would instead use an automatic local variable to obtain the storage and perform initialization. If a pointer were required, use of a smart pointer, such as `std::unique_ptr`, would be a marked improvement. However, these suggested compliant solutions would distract from the lifetime demonstration of this compliant solution and consequently are not shown.

3.5.3 Noncompliant Code Example

In this noncompliant code example, a pointer to an object is implicitly converted to a virtual base class after the object's lifetime has ended, resulting in undefined behavior.

```
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);
}
```

Despite the fact that `f()` never makes use of the object, its being passed as an argument to `f()` is sufficient to trigger undefined behavior.

3.5.4 Compliant Solution

In this compliant solution, the lifetime of `s` is extended to cover the call to `f()`.

```
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;
}
```

3.5.5 Noncompliant Code Example

In this noncompliant code example, the address of a local variable is returned from `f()`. When the resulting pointer is passed to `h()`, the lvalue-to-rvalue conversion applied to `i` results in undefined behavior.

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

void h(int *i);

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

Some compilers generate a diagnostic message when a pointer to an object with automatic storage duration is returned from a function, as in this example.

3.5.6 Compliant Solution

In this compliant solution, the local variable returned from `g()` has static storage duration instead of automatic storage duration, extending its lifetime sufficiently for use within `f()`.

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

void h(int *i);

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

3.5.7 Noncompliant Code Example

A `std::initializer_list<>` object is constructed from an initializer list as though the implementation allocated a temporary array and passed it to the `std::initializer_list<>` constructor. This temporary array has the same lifetime as other temporary objects except that initializing a `std::initializer_list<>` object from the array extends the lifetime of the array exactly like binding a reference to a temporary [ISO/IEC 14882-2014].

In this noncompliant code example, a member variable of type `std::initializer_list<int>` is list-initialized within the constructor's *ctor-initializer*. Under these circumstances, the conceptual temporary array's lifetime ends once the constructor exits, so accessing any elements of the `std::initializer_list<int>` member variable results in undefined behavior.

```
#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();
}
```

3.5.8 Compliant Solution

In this compliant solution, the `std::initializer_list<int>` member variable is replaced with a `std::vector<int>`, which copies the elements of the initializer list to the container instead of relying on a dangling reference to the temporary array.

```
#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();
}
```

3.5.9 Noncompliant Code Example

In this noncompliant code example, a lambda object is stored in a function object, which is later called (executing the lambda) to obtain a constant reference to a value. The lambda object returns an `int` value, which is then stored in a temporary `int` object that becomes bound to the `const int &` return type specified by the function object. However, the temporary object's lifetime is not extended past the return from the function object's invocation, which causes undefined behavior when the resulting value is accessed.

```
#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);
}
```

3.5.10 Compliant Solution

In this compliant solution, the `std::function` object returns an `int` instead of a `const int &`, ensuring that the value is copied instead of bound to a temporary reference. An alternative solution would be to call the lambda directly instead of through the `std::function<>` object.

```
#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);
}
```

3.5.11 Noncompliant Code Example

In this noncompliant code example, the constructor for the automatic variable `s` is not called because execution does not flow through the declaration of the local variable due to the `goto` statement. Because the constructor is not called, the lifetime for `s` has not begun. Therefore, calling `S::f()` uses the object outside of its lifetime and results in undefined behavior.

```
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();
}
```

3.5.12 Compliant Solution

This compliant solution ensures that `s` is properly initialized prior to performing the local jump.

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

void f() {
  S s;

  // ...

  goto bad_idea;

  // ...

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

3.5.13 Noncompliant Code Example

In this noncompliant code example, `f()` is called with an iterable range of objects of type `S`. These objects are copied into a temporary buffer using `std::copy()`, and when processing of those objects is complete, the temporary buffer is deallocated. However, the buffer returned by `std::get_temporary_buffer()` does not contain initialized objects of type `S`, so when `std::copy()` dereferences the destination iterator, it results in undefined behavior because the object referenced by the destination iterator has yet to start its lifetime. This is because while space for the object has been allocated, no constructors or initializers have been invoked.

```
#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);
}
```

3.5.13.1 Implementation Details

A reasonable implementation of `std::get_temporary_buffer()` and `std::copy()` can result in code that behaves like the following example (with error-checking elided).

```
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;
}
```

The act of dereferencing `result` is undefined behavior because the memory pointed to is not an object of type `S` within its lifetime.

3.5.14 Compliant Solution (`std::uninitialized_copy()`)

In this compliant solution, `std::uninitialized_copy()` is used to perform the copy, instead of `std::copy()`, ensuring that the objects are initialized using placement `new` instead of dereferencing uninitialized memory. Identical code from the noncompliant code example has been elided for brevity.

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

3.5.15 Compliant Solution (`std::raw_storage_iterator`)

This compliant solution uses `std::copy()` with a `std::raw_storage_iterator` as the destination iterator with the same well-defined results as using `std::uninitialized_copy()`. As with the previous compliant solution, identical code from the noncompliant code example has been elided for brevity.

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

3.5.16 Risk Assessment

Referencing an object outside of its lifetime can result in an attacker being able to run arbitrary code.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
EXP54-CPP	High	Probable	High	**P6**	**L2**

3.5.17 Related Guidelines

SEI CERT C Coding Standard	DCL30-C. Declare objects with appropriate storage durations

3.5.18 Bibliography

[Coverity 2007]	
[ISO/IEC 14882-2014]	Subclause 3.8, “Object Lifetime” Subclause 8.5.4, “List-Initialization”

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

The C++ Standard, [dcl.type.cv], paragraph 4 [ISO/IEC 14882-2014], states the following:

> Except that any class member declared `mutable` can be modified, any attempt to modify a `const` object during its lifetime results in undefined behavior.

Similarly, paragraph 6 states the following:

> What constitutes an access to an object that has volatile-qualified type is implementation-defined. If an attempt is made to refer to an object defined with a volatile-qualified type through the use of a glvalue with a non-volatile-qualified type, the program behavior is undefined.

Do not cast away a `const` qualification to attempt to modify the resulting object. The `const` qualifier implies that the API designer does not intend for that object to be modified despite the possibility it may be modifiable. Do not cast away a `volatile` qualification; the `volatile` qualifier implies that the API designer intends the object to be accessed in ways unknown to the compiler, and any access of the volatile object results in undefined behavior.

3.6.1 Noncompliant Code Example

In this noncompliant code example, the function `g()` is passed a `const int &`, which is then cast to an `int &` and modified. Because the referenced value was previously declared as `const`, the assignment operation results in undefined behavior.

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

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

3.6.2 Compliant Solution

In this compliant solution, the function `g()` is passed an `int &`, and the caller is required to pass an `int` that can be modified.

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

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

3.6.3 Noncompliant Code Example

In this noncompliant code example, a `const`-qualified method is called that attempts to cache results by casting away the `const`-qualifier of `this`. Because `s` was declared `const`, the mutation of `cachedValue` results in undefined behavior.

```
#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;
}
```

3.6.4 Compliant Solution

This compliant solution uses the `mutable` keyword when declaring `cachedValue`, which allows `cachedValue` to be mutated within a `const` context without triggering undefined behavior.

```
#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;
}
```

3.6.5 Noncompliant Code Example

In this noncompliant code example, the volatile value `s` has the `volatile` qualifier cast away, and an attempt is made to read the value within `g()`, resulting in undefined behavior.

```
#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));
}
```

3.6.6 Compliant Solution

This compliant solution assumes that the volatility of `s` is required, so `g()` is modified to accept a `volatile S &`.

```
#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);
}
```

3.6.7 Exceptions

EXP55-CPP-EX1: An exception to this rule is allowed when it is necessary to cast away `const` when invoking a legacy API that does not accept a `const` argument, provided the function does not attempt to modify the referenced variable. However, it is always preferable to modify the API to be `const`-correct when possible. For example, the following code casts away the `const` qualification of `INVFNAME` in the call to the `audit_log()` function.

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

3.6.8 Risk Assessment

If the object is declared as being constant, it may reside in write-protected memory at runtime. Attempting to modify such an object may lead to abnormal program termination or a denial-of-service attack. If an object is declared as being volatile, the compiler can make no assumptions regarding access of that object. Casting away the volatility of an object can result in reads or writes to the object being reordered or elided entirely, resulting in abnormal program execution.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
EXP55-CPP	Medium	Probable	Medium	**P8**	**L2**

3.6.9 Related Guidelines

SEI CERT C Coding Standard	EXP32-C. Do not access a volatile object through a nonvolatile reference EXP40-C. Do not modify constant objects

3.6.10 Bibliography

[ISO/IEC 14882-2014]	Subclause 7.1.6.1, “The *cv-qualifiers*”
[Sutter 2004]	Item 94, “Avoid Casting Away `const`”

3.7 EXP56-CPP. Do not call a function with a mismatched language linkage

C++ allows a degree of interoperability with other languages through the use of language linkage specifications. These specifications affect the way in which functions are called or data is accessed. By default, all function types, as well as function and variable names, with external linkage have C++ language linkage, though a different language linkage may be specified. Implementations are required to support `"C" and "C++"` as a language linkage, but other language linkages exist with implementation-defined semantics, such as `"java"`, `"Ada"`, and `"FORTRAN"`.

Language linkage is specified to be part of the function type, according to the C++ Standard, [dcl.link], paragraph 1 [ISO/IEC 14882-2014], which, in part, states the following:

> Two function types with different language linkages are distinct types even if they are otherwise identical.

When calling a function, it is undefined behavior if the language linkage of the function type used in the call does not match the language linkage of the function definition. For instance, a mismatch in language linkage specification may corrupt the call stack due to calling conventions or other ABI mismatches.

Do not call a function through a type whose language linkage does not match the language linkage of the called function's definition. This restriction applies both to functions called within a C++ program as well as function pointers used to make a function call from outside of the C++ program.

However, many compilers fail to integrate language linkage into the function's type, despite the normative requirement to do so in the C++ Standard. For instance, GCC 6.1.0, Clang 3.9, and Microsoft Visual Studio 2015 all consider the following code snippet to be ill-formed due to a redefinition of `f()` rather than a well-formed overload of `f()`.

```
typedef void (*cpp_func)(void);
extern "C" typedef void (*c_func)(void);

void f(cpp_func fp) {}
void f(c_func fp) {}
```

Some compilers conform to the C++ Standard, but only in their strictest conformance mode, such as EDG 4.11. This implementation divergence from the C++ Standard is a matter of practical design trade-offs. Compilers are required to support only the `"C"` and `"C++"` language linkages, and interoperability between these two languages often does not require significant code generation differences beyond the mangling of function types for most common architectures such as x86, x86-64, and ARM. There are extant Standard Template Library implementations for which language linkage specifications being correctly implemented as part of the function type would break existing code on common platforms where the language linkage has no effect on the runtime implementation of a function call.

It is acceptable to call a function with a mismatched language linkage when the combination of language linkage specifications, runtime platform, and compiler implementation result in no effect on runtime behavior of the function call. For instance, the following code is permissible when compiled with Microsoft Visual Studio 2015 for x86, despite the lambda function call operator implicitly converting to a function pointer type with C++ language linkage, while `qsort()` expects a function pointer with C language linkage.

```
#include <cstdlib>

void f(int *int_list, size_t count) {
  std::qsort(int_list, count, sizeof(int),
             [](const void *lhs, const void *rhs) -> int {
               return reinterpret_cast<const int *>(lhs) <
                      reinterpret_cast<const int *>(rhs);
             });
}
```

3.7.1 Noncompliant Code Example

In this noncompliant code example, the `call_java_fn_ptr()` function expects to receive a function pointer with `"java"` language linkage because that function pointer will be used by a Java interpreter to call back into the C++ code. However, the function is given a pointer with `"C++"` language linkage instead, resulting in undefined behavior when the interpreter attempts to call the function pointer. This code should be ill-formed because the type of `callback_func()` is different than the type `java_callback`. However, due to common implementation divergence from the C++ Standard, some compilers may incorrectly accept this code without issuing a diagnostic.

```
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);
}
```

3.7.2 Compliant Solution

In this compliant solution, the `callback_func()` function is given `"java"` language linkage to match the language linkage for `java_callback`.

```
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);
}
```

3.7.3 Risk Assessment

Mismatched language linkage specifications generally do not create exploitable security vulnerabilities between the C and C++ language linkages. However, other language linkages exist where the undefined behavior is more likely to result in abnormal program execution, including exploitable vulnerabilities.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
EXP56-CPP	Low	Unlikely	Medium	**P2**	**L3**

3.7.4 Bibliography

[ISO/IEC 14882-2014]	Subclause 5.2.2, “Function Call” Subclause 7.5, “Linkage Specifications”

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

Referring to objects of incomplete class type, also known as *forward declarations*, is a common practice. One such common usage is with the “pimpl idiom” [Sutter 2000] whereby an opaque pointer is used to hide implementation details from a public-facing API. However, attempting to delete a pointer to an object of incomplete class type can lead to undefined behavior. The C++ Standard, [expr.delete], paragraph 5 [ISO/IEC 14882-2014], states the following:

> If the object being deleted has incomplete class type at the point of deletion and the complete class has a non-trivial destructor or a deallocation function, the behavior is undefined.

Do not attempt to delete a pointer to an object of incomplete type. Although it is well-formed if the class has no nontrivial destructor and no associated deallocation function, it would become undefined behavior were a nontrivial destructor or deallocation function added later. It would be possible to check for a nontrivial destructor at compile time using a `static_assert` and the `std::is_trivially_destructible` type trait, but no such type trait exists to test for the presence of a deallocation function.

Pointer downcasting to a pointer of incomplete class type has similar caveats. Pointer upcasting (casting from a more derived type to a less derived type) is a standard implicit conversion operation. C++ allows `static_cast` to perform the inverse operation, pointer downcasting, via [expr.static.cast], paragraph 7. However, when the pointed-to type is incomplete, the compiler is unable to make any class offset adjustments that may be required in the presence of multiple inheritance, resulting in a pointer that cannot be validly dereferenced.

`reinterpret_cast` of a pointer type is defined by [expr.reinterpret.cast], paragraph 7, as being `static_cast<cv T *>(static_cast<cv void *>(PtrValue))`, meaning that `reinterpret_cast` is simply a sequence of `static_cast` operations. C-style casts of a pointer to an incomplete object type are defined as using either `static_cast` or `reinterpret_cast` (it is unspecified which is picked) in [expr.cast], paragraph 5.

Do not attempt to cast through a pointer to an object of incomplete type. The cast operation itself is well-formed, but dereferencing the resulting pointer may result in undefined behavior if the downcast is unable to adjust for multiple inheritance.

3.8.1 Noncompliant Code Example

In this noncompliant code example, a class attempts to implement the pimpl idiom but deletes a pointer to an incomplete class type, resulting in undefined behavior if `Body` has a nontrivial destructor.

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

3.8.2 Compliant Solution (`delete`)

In this compliant solution, the deletion of `impl` is moved to a part of the code where `Body` is defined.

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

// Elsewhere
class Body { /* ... */ };

Handle::~Handle() {
  delete impl;
}
```

3.8.3 Compliant Solution (`std::shared_ptr`)

In this compliant solution, a `std::shared_ptr` is used to own the memory to `impl`. A `std::shared_ptr` is capable of referring to an incomplete type, but a `std::unique_ptr` is not.

```
#include <memory>

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

3.8.4 Noncompliant Code Example

Pointer downcasting (casting a pointer to a base class into a pointer to a derived class) may require adjusting the address of the pointer by a fixed amount that can be determined only when the layout of the class inheritance structure is known. In this noncompliant code example, `f()` retrieves a polymorphic pointer of complete type `B` from `get_d()`. That pointer is then cast to a pointer of incomplete type `D` before being passed to `g()`. Casting to a pointer to the derived class may fail to properly adjust the resulting pointer, causing undefined behavior when the pointer is dereferenced by calling `d->do_something()`.

```
// 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;
}
```

3.8.4.1 Implementation Details

When compiled with Clang 3.8 and the function `f()` is executed, the noncompliant code example prints the following.

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

Similarly, unexpected values are printed when the example is run in Microsoft Visual Studio 2015 and GCC 6.1.0.

3.8.5 Compliant Solution

This compliant solution assumes that the intent is to hide implementation details by using incomplete class types. Instead of requiring a `D *` to be passed to `g()`, it expects a `B *` type.

```
// 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;
}
```

3.8.6 Risk Assessment

Casting pointers or references to incomplete classes can result in bad addresses. Deleting a pointer to an incomplete class results in undefined behavior if the class has a nontrivial destructor. Doing so can cause program termination, a runtime signal, or resource leaks.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
EXP57-CPP	Medium	Unlikely	Medium	**P4**	**L3**

3.8.7 Bibliography

[Dewhurst 2002]	Gotcha #39, "Casting Incomplete Types"
[ISO/IEC 14882-2014]	Subclause 4.10, "Pointer Conversions" Subclause 5.2.9, "Static Cast" Subclause 5.2.10, "Reinterpret Cast" Subclause 5.3.5, "Delete" Subclause 5.4, "Explicit Type Conversion (Cast Notation)"
[Sutter 2000]	"Compiler Firewalls and the Pimpl Idiom"

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

While rule DCL50-CPP. Do not define a C-style variadic function forbids creation of such functions, they may still be defined when that function has external, C-language linkage. Under these circumstances, care must be taken when invoking the `va_start()` macro. The C standard library macro `va_start()` imposes several semantic restrictions on the type of the value of its second parameter. The C Standard, subclause 7.16.1.4, paragraph 4 [ISO/IEC 9899:2011], states the following:

> The parameter *parmN* is the identifier of the rightmost parameter in the variable parameter list in the function definition (the one just before the `...`). If the parameter *parmN* is declared with the `register` storage class, with a function or array type, or with a type that is not compatible with the type that results after application of the default argument promotions, the behavior is undefined.

These restrictions are superseded by the C++ Standard, [support.runtime], paragraph 3 [ISO/IEC 14882-2014], which states the following:

> The restrictions that ISO C places on the second parameter to the `va_start()` macro in header `<stdarg.h>` are different in this International Standard. The parameter `parmN` is the identifier of the rightmost parameter in the variable parameter list of the function definition (the one just before the `...`). If the parameter `parmN` is of a reference type, or of a type that is not compatible with the type that results when passing an argument for which there is no parameter, the behavior is undefined.

The primary differences between the semantic requirements are as follows:

- You must not pass a reference as the second argument to `va_start()`.
- Passing an object of a class type that has a nontrivial copy constructor, nontrivial move constructor, or nontrivial destructor as the second argument to `va_start` is conditionally supported with implementation-defined semantics ([expr.call] paragraph 7).
- You may pass a parameter declared with the `register` keyword ([dcl.stc] paragraph 3) or a parameter with a function type.

Passing an object of array type still produces undefined behavior in C++ because an array type as a function parameter requires the use of a reference, which is prohibited. Additionally, passing an object of a type that undergoes default argument promotions still produces undefined behavior in C++.

3.9.1 Noncompliant Code Example

In this noncompliant code example, the object passed to `va_start()` will undergo a default argument promotion, which results in undefined behavior.

```
#include <cstdarg>

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

3.9.2 Compliant Solution

In this compliant solution, `f()` accepts a `double` instead of a `float`.

```
#include <cstdarg>

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

3.9.3 Noncompliant Code Example

In this noncompliant code example, a reference type is passed as the second argument to `va_start()`.

```
#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);
}
```

3.9.4 Compliant Solution

Instead of passing a reference type to `f()`, this compliant solution passes a pointer type.

```
#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);
}
```

3.9.5 Noncompliant Code Example

In this noncompliant code example, a class with a nontrivial copy constructor (`std::string`) is passed as the second argument to `va_start()`, which is conditionally supported depending on the implementation.

```
#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);
}
```

3.9.6 Compliant Solution

This compliant solution passes a `const char *` instead of a `std::string`, which has well-defined behavior on all implementations.

```
#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);
}
```

3.9.7 Risk Assessment

Passing an object of an unsupported type as the second argument to `va_start()` can result in undefined behavior that might be exploited to cause data integrity violations.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
EXP58-CPP	Medium	Unlikely	Medium	**P4**	**L3e**

3.9.8 Related Guidelines

SEI CERT C++ Coding Standard	DCL50-CPP. Do not define a C-style variadic function

3.9.9 Bibliography

[ISO/IEC 9899:2011]	Subclause 7.16.1.4, “The `va_start` Macro”
[ISO/IEC 14882-2014]	Subclause 18.10, “Other Runtime Support”

3.10 EXP59-CPP. Use offsetof() on valid types and members

The `offsetof()` macro is defined by the C Standard as a portable way to determine the offset, expressed in bytes, from the start of the object to a given member of that object. The C Standard, subclause 7.17, paragraph 3 [ISO/IEC 9899:1999], in part, specifies the following:

> `offsetof(type, member-designator)` which expands to an integer constant expression that has type `size_t`, the value of which is the offset in bytes, to the structure member (designated by *member-designator*), from the beginning of its structure (designated by *type*). The type and member designator shall be such that given `static type t;` then the expression `&(t.member-designator)` evaluates to an address constant. (If the specified member is a bit-field, the behavior is undefined.)

The C++ Standard, [support.types], paragraph 4 [ISO/IEC 14882-2014], places additional restrictions beyond those set by the C Standard:

> The macro `offsetof(type, member-designator)` accepts a restricted set of *type* arguments in this International Standard. If *type* is not a *standard-layout class*, the results are undefined. The expression `offsetof(type, member-designator)` is never type-dependent and it is value-dependent if and only if *type* is dependent. The result of applying the `offsetof` macro to a field that is a static data member or a function member is undefined. No operation invoked by the `offsetof` macro shall throw an exception and `noexcept(offsetof(type, member-designator))` shall be true.

When specifying the type argument for the `offsetof()` macro, pass only a standard-layout class. The full description of a standard-layout class can be found in paragraph 7 of the [class] clause of the C++ Standard, or the type can be checked with the `std::is_standard_layout<>` type trait. When specifying the member designator argument for the `offsetof()` macro, do not pass a bit-field, static data member, or function member. Passing an invalid type or member to the `offsetof()` macro is undefined behavior.

3.10.1 Noncompliant Code Example

In this noncompliant code example, a type that is not a standard-layout class is passed to the `offsetof()` macro, resulting in undefined behavior.

```
#include <cstddef>

struct D {
  virtual void f() {}
  int i;
};

void f() {
  size_t off = offsetof(D, i);
  // ...
}
```

3.10.1.1 Implementation Details

The noncompliant code example does not emit a diagnostic when compiled with the `/Wall` switch in Microsoft Visual Studio 2015 on x86, resulting in `off` being `4`, due to the presence of a vtable for type `D`.

3.10.2 Compliant Solution

It is not possible to determine the offset to `i` within `D` because `D` is not a standard-layout class. However, it is possible to make a standard-layout class within `D` if this functionality is critical to the application, as demonstrated by this compliant solution.

```
#include <cstddef>

struct D {
  virtual void f() {}
  struct InnerStandardLayout {
    int i;
  } inner;
};

void f() {
  size_t off = offsetof(D::InnerStandardLayout, i);
  // ...
}
```

3.10.3 Noncompliant Code Example

In this noncompliant code example, the offset to `i` is calculated so that a value can be stored at that offset within `buffer`. However, because `i` is a static data member of the class, this example results in undefined behavior. According to the C++ Standard, [class.static.data], paragraph 1 [ISO/IEC 14882-2014], static data members are not part of the subobjects of a class.

```
#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);
}
```

3.10.3.1 Implementation Details

The noncompliant code example does not emit a diagnostic when compiled with the `/Wall` switch in Microsoft Visual Studio 2015 on x86, resulting in `off` being a large value representing the offset between the null pointer address `0` and the address of the static variable `S::i`.

3.10.4 Compliant Solution

Because static data members are not a part of the class layout, but are instead an entity of their own, this compliant solution passes the address of the static member variable as the buffer to store the data in and passes `0` as the offset.

```
#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);
}
```

3.10.5 Risk Assessment

Passing an invalid type or member to `offsetof()` can result in undefined behavior that might be exploited to cause data integrity violations or result in incorrect values from the macro expansion.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
EXP59-CPP	Medium	Unlikely	Medium	**P4**	**L3**

3.10.6 Bibliography

[ISO/IEC 9899:1999]	Subclause 7.17, “Common Definitions `<stddef.h>`”
[ISO/IEC 14882-2014]	Subclause 9.4.2, “Static Data Members” Subclause 18.2, “Types”

3.11 EXP60-CPP. Do not pass a nonstandard-layout type object across execution boundaries

Standard-layout types can be used to communicate with code written in other programming languages, as the layout of the type is strictly specified. The C++ Standard, [class], paragraph 7 [ISO/IEC 14882-2014], defines a standard-layout class as a class that

- does not have virtual functions,
- has the same access control for all nonstatic data members,
- has no base classes of the same type as the first nonstatic data member,
- has nonstatic data members declared in only one class within the class hierarchy, and
- recursively, does not have nonstatic data members of nonstandard-layout type.

An *execution boundary* is the delimitation between code compiled by differing compilers, including different versions of a compiler produced by the same vendor. For instance, a function may be declared in a header file but defined in a library that is loaded at runtime. The execution boundary exists between the call site in the executable and the function implementation in the library. Such boundaries are also called ABI (application binary interface) boundaries because they relate to the interoperability of application binaries.

Do not make any assumptions about the specific layout of objects with nonstandard-layout types. For objects compiled by one compiler that are referenced by code compiled by a different compiler, such assumptions cause correctness and portability concerns. The layout of the object generated by the first compiler is not guaranteed to be identical to the layout generated by the second compiler, even if both compilers are conforming C++ implementations. However, some implementations may document binary compatibility guarantees that can be relied on for passing nonstandard-layout objects between execution boundaries.

A special instance of this guidance involves non-C++ code compiled by a different compiler, such as C standard library implementations that are exposed via the C++ standard library. C standard library functions are exposed with C++ signatures, and the type system frequently assists in ensuring that types match appropriately. This process disallows passing a pointer to a C++ object to a function expecting a `char *` without additional work to suppress the type mismatch. However, some C standard library functions accept a `void *` for which any C++ pointer type will suffice. Passing a pointer to a nonstandard-layout type in this situation results in indeterminate behavior because it depends on the behavior of the other language as well as on the layout of the given object. For more information, see rule EXP56-CPP. Do not call a function with a mismatched language linkage.

Pass a nonstandard-layout type object across execution boundaries only when both sides of the execution boundary adhere to the same ABI. This is permissible if the same version of a compiler is used to compile both sides of the execution boundary, if the compiler used to compile both sides of the execution boundary is ABI-compatible across multiple versions, or if the differing compilers document that they adhere to the same ABI.

3.11.1 Noncompliant Code Example

This noncompliant code example assumes that there is a library whose header is `library.h`, an application (represented by `application.cpp`), and that the library and application are not ABI-compatible. Therefore, the contents of `library.h` constitute an execution boundary. A nonstandard-layout type object `S` is passed across this execution boundary. The application creates an instance of an object of this type, then passes a reference to the object to a function defined by the library, crossing the execution boundary. Because the layout is not guaranteed to be compatible across the boundary, this results in unexpected behavior.

```
// 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);
}
```

This example would be compliant if the library and the application conformed to the same ABI, either explicitly through vendor documentation or implicitly by virtue of using the same compiler version to compile both.

3.11.2 Compliant Solution

Because the library and application do not conform to the same ABI, this compliant solution modifies the library and application to work with a standard-layout type. Furthermore, it also adds a `static_assert()` to help guard against future code changes that accidentally modify `S` to no longer be a standard-layout type.

```
// 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);
}
```

3.11.3 Noncompliant Code Example

In this noncompliant code example, a pointer to an object of nonstandard-layout type is passed to a function that has a `"Fortran"` language linkage. Language linkages other than `"C"` and `"C++"` are conditionally supported with implementation-defined semantics [ISO/IEC 14882-2014]. If the implementation does not support this language linkage, the code is ill-formed. Assuming that the language linkage is supported, any operations performed on the object passed may result in indeterminate behavior, which could have security implications.

```
struct B {
  int i, j;
};

struct D : B {
  float f;
};

extern "Fortran" void func(void *);

void foo(D *d) {
  func(d);
}
```

3.11.4 Compliant Solution

In this compliant solution, the nonstandard-layout type object is serialized into a local standard-layout type object, which is then passed to the `Fortran` function.

```
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;

  func(&temp);
}
```

3.11.5 Risk Assessment

The effects of passing objects of nonstandard-layout type across execution boundaries depends on what operations are performed on the object within the callee as well as what subsequent operations are performed on the object from the caller. The effects can range from correct or benign behavior to undefined behavior.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
EXP60-CPP	High	Probable	Medium	**P12**	**L1**

3.11.6 Related Guidelines

CERT C++ Coding Standard	EXP58-CPP. Pass an object of the correct type to va_start

3.11.7 Bibliography

[ISO/IEC 14882-2014]	Clause 9, "Classes" Subclause 7.5, "Linkage Specifications"

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

Lambda expressions may capture objects with automatic storage duration from the set of enclosing scopes (called the *reaching scope*) for use in the lambda's function body. These captures may be either explicit, by specifying the object to capture in the lambda's *capture-list*, or implicit, by using a *capture-default* and referring to the object within the lambda's function body. When capturing an object explicitly or implicitly, the capture-default indicates that the object is either captured by copy (using `=`) or captured by reference (using `&`). When an object is captured by copy, the lambda object will contain an unnamed nonstatic data member that is initialized to the value of the object being captured. This nonstatic data member's lifetime is that of the lambda object's lifetime. However, when an object is captured by reference, the lifetime of the referent is not tied to the lifetime of the lambda object.

Because entities captured are objects with automatic storage duration (or `this`), a general guideline is that functions returning a lambda object (including returning via a reference parameter), or storing a lambda object in a member variable or global, should not capture an entity by reference because the lambda object often outlives the captured reference object.

When a lambda object outlives one of its reference-captured objects, execution of the lambda object's function call operator results in undefined behavior once that reference-captured object is accessed. Therefore, a lambda object must not outlive any of its reference-captured objects. This is a specific instance of EXP54-CPP. Do not access an object outside of its lifetime.

3.12.1 Noncompliant Code Example

In this noncompliant code example, the function `g()` returns a lambda, which implicitly captures the automatic local variable `i` by reference. When that lambda is returned from the call, the reference it captured will refer to a variable whose lifetime has ended. As a result, when the lambda is executed in `f()`, the use of the dangling reference in the lambda results in undefined behavior.

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

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

3.12.2 Compliant Solution

In this compliant solution, the lambda does not capture `i` by reference but instead captures it by copy. Consequently, the lambda contains an implicit nonstatic data member whose lifetime is that of the lambda.

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

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

3.12.3 Noncompliant Code Example

In this noncompliant code example, a lambda reference captures a local variable from an outer lambda. However, this inner lambda outlives the lifetime of the outer lambda and any automatic local variables it defines, resulting in undefined behavior when an inner lambda object is executed within `f()`.

```
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();
}
```

3.12.4 Compliant Solution

In this compliant solution, the inner lambda captures `i` by copy instead of by reference.

```
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();
}
```

3.12.5 Risk Assessment

Referencing an object outside of its lifetime can result in an attacker being able to run arbitrary code.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
EXP61-CPP	High	Probable	High	**P6**	**L2**

3.12.6 Related Guidelines

SEI CERT C++ Coding Standard	EXP54-CPP. Do not access an object outside of its lifetime

3.12.7 Bibliography

[ISO/IEC 14882-2014]	Subclause 3.8, “Object Lifetime” Subclause 5.1.2, “Lambda Expressions”

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

The C++ Standard, [basic.types], paragraph 9 [ISO/IEC 14882-2014], states the following:

> The *object representation* of an object of type `T` is the sequence of *N* `unsigned char` objects taken up by the object of type `T`, where *N* equals `sizeof(T)`. The *value representation* of an object is the set of bits that hold the value of type `T`.

The narrow character types (`char`, `signed char`, and `unsigned char`)—as well as some other integral types on specific platforms—have an object representation that consists solely of the bits from the object’s value representation. For such types, accessing any of the bits of the value representation is well-defined behavior. This form of object representation allows a programmer to access and modify an object solely based on its bit representation, such as by calling `std::memcmp()` on its object representation.

Other types, such as classes, may not have an object representation composed solely of the bits from the object’s value representation. For instance, classes may have bit-field data members, padding inserted between data members, a vtable to support virtual method dispatch, or data members declared with different access privileges. For such types, accessing bits of the object representation that are not part of the object’s value representation may result in undefined behavior depending on how those bits are accessed.

Do not access the bits of an object representation that are not part of the object’s value representation. Even if the bits are accessed in a well-defined manner, such as through an array of `unsigned char` objects, the values represented by those bits are unspecified or implementation-defined, and reliance on any particular value can lead to abnormal program execution.

3.13.1 Noncompliant Code Example

In this noncompliant code example, the complete object representation is accessed when comparing two objects of type S. Per the C++ Standard, [class], paragraph 13 [ISO/IEC 14882-2014], classes may be padded with data to ensure that they are properly aligned in memory. The contents of the padding and the amount of padding added is implementation-defined. This can lead to incorrect results when comparing the object representation of classes instead of the value representation, as the padding may assume different unspecified values for each object instance.

```
#include <cstring>

struct S {
  unsigned char buffType;
  int size;
};

void f(const S &s1, const S &s2) {
  if (!std::memcmp(&s1, &s2, sizeof(S))) {
    // ...
  }
}
```

3.13.2 Compliant Solution

In this compliant solution, S overloads `operator==()` to perform a comparison of the value representation of the object.

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

3.13.3 Noncompliant Code Example

In this noncompliant code example, `std::memset()` is used to clear the internal state of an object. An implementation may store a vtable within the object instance due to the presence of a virtual function, and that vtable is subsequently overwritten by the call to `std::memset()`, leading to undefined behavior when virtual method dispatch is required.

```
#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
}
```

3.13.4 Compliant Solution

In this compliant solution, the data members of `S` are cleared explicitly instead of calling `std::memset()`.

```
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
}
```

3.13.5 Exceptions

EXP62-CPP-EX1: It is permissible to access the bits of an object representation when that access is otherwise unobservable in well-defined code. Specifically, reading bits that are not part of the value representation is permissible when there is no reliance or assumptions placed on their values, and writing bits that are not part of the value representation is only permissible when those bits are padding bits. This exception does not permit writing to bits that are part of the object representation aside from padding bits, such as overwriting a vtable pointer.

For instance, it is acceptable to call `std::memcpy()` on an object containing a bit-field, as in the following example, because the read and write of the padding bits cannot be observed.

```
#include <cstring>

struct S {
  int i : 10;
  int j;
};

void f(const S &s1) {
  S s2;
  std::memcpy(&s2, &s1, sizeof(S));
}
```

Code that complies with this exception must still comply with OOP57-CPP. Prefer special member functions and overloaded operators to C Standard Library functions.

3.13.6 Risk Assessment

The effects of accessing bits of an object representation that are not part of the object's value representation can range from implementation-defined behavior (such as assuming the layout of fields with differing access controls) to code execution vulnerabilities (such as overwriting the vtable pointer).

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
EXP62-CPP	High	Probable	High	**P6**	**L2**

3.13.7 Related Guidelines

SEI CERT C++ Coding Standard	OOP57-CPP. Prefer special member functions and overloaded operators to C Standard Library functions

3.13.8 Bibliography

[ISO/IEC 14882-2014]	Subclause 3.9, “Types” Subclause 3.10, “Lvalues and Rvalues” Clause 9, “Classes”

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

Many types, including user-defined types and types provided by the Standard Template Library, support move semantics. Except in rare circumstances, an object of a type that supports move operations (move initialization or move assignment) will be left in a valid, but unspecified state after the object's value has been moved.

Passing an object as a function argument that binds to an rvalue reference parameter, including via implicit function call syntax such as move assignment or move construction, *moves* the object's state into another object. Upon return from such a function call, the object that was bound as an rvalue reference parameter is considered to be in the *moved-from* state. Once an object is in the moved-from state, the only operations that may be safely performed on that object instance are ones for which the operation has no preconditions, because it is unknown whether the unspecified state of the object will satisfy those preconditions. While some types have explicitly-defined preconditions, such as types defined by the Standard Template Library, it should be assumed that the only operations that may be safely performed on a moved-from object instance are reinitialization through assignment into the object or terminating the lifetime of the object by invoking its destructor.

Do not rely on the value of a moved-from object unless the type of the object is documented to be in a well-specified state. While the object is guaranteed to be in a valid state, relying on unspecified values leads to unspecified behavior. Since the behavior need not be documented, this can in turn result in abnormal program behavior and portability concerns.

The following Standard Template Library functions are guaranteed to leave the moved-from object in a well-specified state.

Type	Functionality	Moved-from State
`std::unique_ptr`	Move construction, Move assignment, "Converting" move construction, "Converting" move assignment (likewise for `std::unique_ptr` for array objects with a runtime length)	The moved-from object is guaranteed to refer to a null pointer value, per [unique.ptr], paragraph 4 [ISO/IEC 14882-2014].
`std::shared_ptr`	Move construction, Move assignment, "Converting" move construction, "Converting" move assignment	The moved-from object shall be "empty," per [util.smartptr.shared.const], paragraph 22 and [util.smartptr.shared.assign], paragraph 4.
`std::shared_ptr`	Move construction, Move assignment from a `std::unique_ptr`	The moved-from object is guaranteed to refer to a null pointer value, per [util.smartptr.shared.const], paragraph 29 and [util.smartptr.shared.assign], paragraph 6.

Type	Functionality	Moved-from State
`std::weak_ptr`	Move construction, Move assignment, "Converting" move construction, "Converting" move assignment	The moved-from object shall be "empty," per [util.smartptr.weak.const], paragraph 8, and [util.smartptr.weak.assign] , paragraph 4.
`std::basic_ios`	`move()`	The moved-from object is still left in an unspecified state, except that `rdbuf()` shall return the same value as it returned before the move, and `tie()` shall return `0`, per [basic.ios.members], paragraph 20.
`std::basic_filebuf`	Move constructor, Move assignment	The moved-from object is guaranteed to reference no file; other internal state is also affected, per [filebuf.cons], paragraphs 3 and 4, and [filebuf.assign], paragraph 1.
`std::thread`	Move constructor, Move assignment	The result from calling `get_id()` on the moved-from object is guaranteed to remain unchanged; otherwise the object is in an unspecified state, per [thread.thread.constr], paragraph 11 and [thread.thread.assign], paragraph 2.
`std::unique_lock`	Move constructor, Move assignment	The moved-from object is guaranteed to be in its default state, per [thread.lock.unique.cons], paragraphs 21 and 23.
`std::shared_lock`	Move constructor, Move assignment	The moved-from object is guaranteed to be in its default state, per [thread.lock.shared.cons], paragraphs 21 and 23.
`std::promise`	Move constructor, Move assignment	The moved-from object is guaranteed not to have any shared state, per [futures.promise], paragraphs 6 and 8.
`std::future`	Move constructor, Move assignment	Calling `valid()` on the moved-from object is guaranteed to return `false`, per [futures.unique_future], paragraphs 8 and 11.
`std::shared_future`	Move constructor, Move assignment, "Converting" move constructor, "Converting" move assignment	Calling `valid()` on the moved-from object is guaranteed to return `false`, per [futures.shared_future], paragraphs 8 and 11.
`std::packaged_task`	Move constructor, Move assignment	The moved-from object is guaranteed not to have any shared state, per [future.task.members], paragraphs 7 and 8.

Several generic standard template library (STL) algorithms, such as `std::remove()` and `std::unique()`, remove instances of elements from a container without shrinking the size of the container. Instead, these algorithms return a `ForwardIterator` to indicate the partition within the container after which elements are no longer valid. The elements in the container that precede the returned iterator are valid elements with specified values; whereas the elements that succeed the returned iterator are valid but have unspecified values. Accessing unspecified values of elements iterated over results in unspecified behavior. Frequently, the erase-remove idiom is used to shrink the size of the container when using these algorithms.

3.14.1 Noncompliant Code Example

In this noncompliant code example, the integer values `0` through `9` are expected to be printed to the standard output stream from a `std::string` rvalue reference. However, because the object is moved and then reused under the assumption its internal state has been cleared, unexpected output may occur despite not triggering undefined behavior.

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

3.14.1.1 Implementation Details

Some standard library implementations may implement the *short string optimization (SSO)* when implementing `std::string`. In such implementations, strings under a certain length are stored in a character buffer internal to the `std::string` object (avoiding an expensive heap allocation operation). However, such an implementation might not alter the original buffer value when performing a move operation. When the noncompliant code example is compiled with Clang 3.7 using libc++, the following output is produced.

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

3.14.2 Compliant Solution

In this compliant solution, the `std::string` object is initialized to the expected value on each iteration of the loop. This practice ensures that the object is in a valid, specified state prior to attempting to access it in `g()`, resulting in the expected output.

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

3.14.3 Noncompliant Code Example

In this noncompliant code example, elements matching `42` are removed from the given container. The contents of the container are then printed to the standard output stream. However, if any elements were removed from the container, the range-based `for` loop iterates over an invalid iterator range, resulting in unspecified behavior.

```
#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;
  }
}
```

3.14.4 Compliant Solution

In this compliant solution, elements removed by the standard algorithm are skipped during iteration.

```
#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;
    }
  }
}
```

3.14.5 Compliant Solution

In this compliant solution, elements removed by the standard algorithm are subsequently erased from the given container. This technique ensures that a valid iterator range is used by the range-based `for` loop.

```
#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;
  }
}
```

3.14.6 Risk Assessment

The state of a moved-from object is generally valid, but unspecified. Relying on unspecified values can lead to abnormal program termination as well as data integrity violations.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
EXP63-CPP	Medium	Probable	Medium	**P8**	**L2**

3.14.7 Bibliography

[ISO/IEC 14882-2014]	Subclause 17.6.5.15, “Moved-from State of Library Types” Subclause 20.8.1, “Class Template `unique_ptr`” Subclause 20.8.2, “Shared-Ownership Pointers” Subclause 27.5.5, “Class Template `basic_ios`” Subclause 27.9.1, “File Streams” Subclause 30.3.1, “Class `thread`” Subclause 30.4.2, “Locks” Subclause 30.6, “Futures”

4 Integers (INT)

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

Enumerations in C++ come in two forms: *scoped* enumerations in which the underlying type is fixed and *unscoped* enumerations in which the underlying type may or may not be fixed. The range of values that can be represented by either form of enumeration may include enumerator values not specified by the enumeration itself. The range of valid enumeration values for an enumeration type is defined by the C++ Standard, [dcl.enum], in paragraph 8 [ISO/IEC 14882-2014]:

> For an enumeration whose underlying type is fixed, the values of the enumeration are the values of the underlying type. Otherwise, for an enumeration where e_{min} is the smallest enumerator and e_{max} is the largest, the values of the enumeration are the values in the range b_{min} to b_{max}, defined as follows: Let K be `1` for a two's complement representation and `0` for a one's complement or sign-magnitude representation. b_{max} is the smallest value greater than or equal to $max(|e_{min}| - K, |e_{max}|)$ and equal to $2^M - 1$, where M is a non-negative integer. b_{min} is zero if e_{min} is non-negative and $-(b_{max} + K)$ otherwise. The size of the smallest bit-field large enough to hold all the values of the enumeration type is $max(M, 1)$ if b_{min} is zero and $M + 1$ otherwise. It is possible to define an enumeration that has values not defined by any of its enumerators. If the enumerator-list is empty, the values of the enumeration are as if the enumeration had a single enumerator with value `0`.

The C++ Standard, [expr.static.cast], paragraph 10, states the following:

> A value of integral or enumeration type can be explicitly converted to an enumeration type. The value is unchanged if the original value is within the range of the enumeration values (7.2). Otherwise, the resulting value is unspecified (and might not be in that range). A value of floating-point type can also be explicitly converted to an enumeration type. The resulting value is the same as converting the original value to the underlying type of the enumeration (4.9), and subsequently to the enumeration type.

To avoid operating on unspecified values, the arithmetic value being cast must be within the range of values the enumeration can represent. When dynamically checking for out-of-range values, checking must be performed before the cast expression.

4.1.1 Noncompliant Code Example (Bounds Checking)

This noncompliant code example attempts to check whether a given value is within the range of acceptable enumeration values. However, it is doing so after casting to the enumeration type, which may not be able to represent the given integer value. On a two's complement system, the valid range of values that can be represented by `EnumType` are [0..3], so if a value outside of that range were passed to `f()`, the cast to `EnumType` would result in an unspecified value, and using that value within the `if` statement results in unspecified behavior.

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

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

  if (enumVar < First || enumVar > Third) {
    // Handle error
  }
}
```

4.1.2 Compliant Solution (Bounds Checking)

This compliant solution checks that the value can be represented by the enumeration type before performing the conversion to guarantee the conversion does not result in an unspecified value. It does this by restricting the converted value to one for which there is a specific enumerator value.

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

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

4.1.3 Compliant Solution (Scoped Enumeration)

This compliant solution uses a scoped enumeration, which has a fixed underlying `int` type by default, allowing any value from the parameter to be converted into a valid enumeration value. It does not restrict the converted value to one for which there is a specific enumerator value, but it could do so as shown in the previous compliant solution.

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

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

4.1.4 Compliant Solution (Fixed Unscoped Enumeration)

Similar to the previous compliant solution, this compliant solution uses an unscoped enumeration but provides a fixed underlying `int` type allowing any value from the parameter to be converted to a valid enumeration value.

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

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

Although similar to the previous compliant solution, this compliant solution differs from the noncompliant code example in the way the enumerator values are expressed in code and which implicit conversions are allowed. The previous compliant solution requires a nested name specifier to identify the enumerator (for example, `EnumType::First`) and will not implicitly convert the enumerator value to `int`. As with the noncompliant code example, this compliant solution does not allow a nested name specifier and will implicitly convert the enumerator value to `int`.

4.1.5 Risk Assessment

It is possible for unspecified values to result in a buffer overflow, leading to the execution of arbitrary code by an attacker. However, because enumerators are rarely used for indexing into arrays or other forms of pointer arithmetic, it is more likely that this scenario will result in data integrity violations rather than arbitrary code execution.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
INT50-CPP	Medium	Unlikely	Medium	**P4**	**L3**

4.1.6 Bibliography

[Becker 2009]	Section 7.2, “Enumeration Declarations”
[ISO/IEC 14882-2014]	Subclause 5.2.9, “Static Cast” Subclause 7.2, “Enumeration Declarations”

5 Containers (CTR)

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

Ensuring that array references are within the bounds of the array is almost entirely the responsibility of the programmer. Likewise, when using standard template library vectors, the programmer is responsible for ensuring integer indexes are within the bounds of the vector.

5.1.1 Noncompliant Code Example (Pointers)

This noncompliant code example shows a function, `insert_in_table()`, that has two `int` parameters, `pos` and `value`, both of which can be influenced by data originating from untrusted sources. The function performs a range check to ensure that `pos` does not exceed the upper bound of the array, specified by `tableSize`, but fails to check the lower bound. Because `pos` is declared as a (signed) `int`, this parameter can assume a negative value, resulting in a write outside the bounds of the memory referenced by `table`.

```
#include <cstddef>

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

5.1.2 Compliant Solution (`size_t`)

In this compliant solution, the parameter `pos` is declared as `size_t`, which prevents the passing of negative arguments.

```
#include <cstddef>

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

5.1.3 Compliant Solution (Non-Type Templates)

Non-type templates can be used to define functions accepting an array type where the array bounds are deduced at compile time. This compliant solution is functionally equivalent to the previous bounds-checking one except that it additionally supports calling `insert_in_table()` with an array of known bounds.

```
#include <cstddef>
#include <new>

void insert_in_table(int *table, std::size_t tableSize, std::size_t
pos, int value) {
  // #1
  if (pos >= tableSize) {
    // Handle error
    return;
  }
  table[pos] = value;
}

template <std::size_t N>
void insert_in_table(int (&table)[N], std::size_t pos, int value) {
  // #2
  insert_in_table(table, N, pos, value);
}

void f() {
  // Exposition only
  int table1[100];
  int *table2 = new int[100];
  insert_in_table(table1, 0, 0); // Calls #2
  insert_in_table(table2, 0, 0); // Error, no matching func. call
  insert_in_table(table1, 100, 0, 0); // Calls #1
  insert_in_table(table2, 100, 0, 0); // Calls #1
  delete [] table2;
}
```

5.1.4 Noncompliant Code Example (`std::vector`)

In this noncompliant code example, a `std::vector` is used in place of a pointer and size pair. The function performs a range check to ensure that `pos` does not exceed the upper bound of the container. Because `pos` is declared as a (signed) `long long`, this parameter can assume a negative value. On systems where `std::vector::size_type` is ultimately implemented as an `unsigned int` (such as with Microsoft Visual Studio 2013), the usual arithmetic conversions applied for the comparison expression will convert the unsigned value to a signed value. If `pos` has a negative value, this comparison will not fail, resulting in a write outside the bounds of the `std::vector` object when the negative value is interpreted as a large unsigned value in the indexing operator.

```
#include <vector>

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

5.1.5 Compliant Solution (`std::vector`, `size_t`)

In this compliant solution, the parameter `pos` is declared as `size_t`, which ensures that the comparison expression will fail when a large, positive value (converted from a negative argument) is given.

```
#include <vector>

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

5.1.6 Compliant Solution (`std::vector::at()`)

In this compliant solution, access to the vector is accomplished with the `at()` method. This method provides bounds checking, throwing a `std::out_of_range` exception if `pos` is not a valid index value. The `insert_in_table()` function is declared with `noexcept(false)` in compliance with ERR55-CPP. Honor exception specifications.

```
#include <vector>

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

5.1.7 Noncompliant Code Example (Iterators)

In this noncompliant code example, the `f_imp()` function is given the (correct) ending iterator `e` for a container, and `b` is an iterator from the same container. However, it is possible that `b` is not within the valid range of its container. For instance, if the container were empty, `b` would equal `e` and be improperly dereferenced.

```
#include <iterator>

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

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

5.1.8 Compliant Solution

This compliant solution tests for iterator validity before attempting to dereference `b`.

```
#include <iterator>

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

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

5.1.9 Risk Assessment

Using an invalid array or container index can result in an arbitrary memory overwrite or abnormal program termination.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
CTR50-CPP	High	Likely	High	**P9**	**L2**

5.1.10 Related Guidelines

SEI CERT C Coding Standard	ARR30-C. Do not form or use out-of-bounds pointers or array subscripts
MITRE CWE	CWE 119, Failure to Constrain Operations within the Bounds of a Memory Buffer CWE 129, Improper Validation of Array Index

5.1.11 Bibliography

[ISO/IEC 14882-2014]	Clause 23, “Containers Library” Subclause 24.2.1, “In General”
[ISO/IEC TR 24772-2013]	Boundary Beginning Violation [XYX] Wrap-Around Error [XYY] Unchecked Array Indexing [XYZ]
[Viega 2005]	Section 5.2.13, “Unchecked Array Indexing”

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

Iterators are a generalization of pointers that allow a C++ program to work with different data structures (containers) in a uniform manner [ISO/IEC 14882-2014]. Pointers, references, and iterators share a close relationship in which it is required that referencing values be done through a valid iterator, pointer, or reference. Storing an iterator, reference, or pointer to an element within a container for any length of time comes with a risk that the underlying container may be modified such that the stored iterator, pointer, or reference becomes invalid. For instance, when a sequence container such as `std::vector` requires an underlying reallocation, outstanding iterators, pointers, and references will be invalidated [Kalev 1999]. Use only a valid pointer, reference, or iterator to refer to an element of a container.

The C++ Standard, [container.requirements.general], paragraph 12 [ISO/IEC 14882-2014] states the following:

> Unless otherwise specified (either explicitly or by defining a function in terms of other functions), invoking a container member function or passing a container as an argument to a library function shall not invalidate iterators to, or change the values of, objects within that container.

The C++ Standard allows references and pointers to be invalidated independently for the same operation, which may result in an invalidated reference but not an invalidated pointer. However, relying on this distinction is insecure because the object pointed to by the pointer may be different than expected even if the pointer is valid. For instance, it is possible to retrieve a pointer to an element from a container, erase that element (invalidating references when destroying the underlying object), then insert a new element at the same location within the container causing the extant pointer to now point to a valid, but distinct object. Thus, any operation that invalidates a pointer or a reference should be treated as though it invalidates both pointers and references.

The following container functions can invalidate iterators, references, and pointers under certain circumstances.

Class	Function	Iterators	References/ Pointers	Notes
`std::deque`				
	`insert()`, `emplace_front()`, `emplace_back()`, `emplace()`, `push_front()`, `push_back()`,	X	X	An insertion in the middle of the deque invalidates all the iterators and references to elements of the deque. An insertion at either end of the deque invalidates all the iterators to the deque but has no effect on the validity of references to elements of the deque. ([deque.modifiers], paragraph 1)
	`erase()`, `pop_back()`, `resize()`	X	X	An erase operation that erases the last element of a deque invalidates only the past-the-end iterator and all iterators and references to the erased elements. An erase operation that erases the first element of a deque but not the last element invalidates only the erased elements. An erase operation that erases neither the first element nor the last element of a deque invalidates the past-the-end iterator and all iterators and references to all the elements of the deque. ([deque.modifiers], paragraph 4)
	`clear()`	X	X	Destroys all elements in the container. Invalidates all references, pointers, and iterators referring to the elements of the container and may invalidate the past-the-end iterator. ([sequence.reqmts], Table 100)
`std::forward_list`				

Class	Function	Iterators	References/ Pointers	Notes
	`erase_after()`, `pop_front()`, `resize()`	X	X	erase_after shall invalidate only iterators and references to the erased elements. ([forwardlist.modifiers], paragraph 1)
	`remove()`, `unique()`	X	X	Invalidates only the iterators and references to the erased elements. ([forwardlist.ops], paragraph 12 & paragraph 16)
	`clear()`	X	X	Destroys all elements in the container. Invalidates all references, pointers, and iterators referring to the elements of the container and may invalidate the past-the-end iterator. ([sequence.reqmts], Table 100)
`std::list`				
	`erase()`, `pop_front()`, `pop_back()`, `clear()`, `remove()`, `remove_if()`, `unique()`	X	X	Invalidates only the iterators and references to the erased elements. ([list.modifiers], paragraph 3 and [list.ops], paragraph 15 & paragraph 19)
	`clear()`	X	X	Destroys all elements in the container. Invalidates all references, pointers, and iterators referring to the elements of the container and may invalidate the past-the-end iterator. ([sequence.reqmts], Table 100)
`std::vector`				

Class	Function	Iterators	References/ Pointers	Notes
	`reserve()`	X	X	After `reserve()`, `capacity()` is greater or equal to the argument of `reserve` if reallocation happens and is equal to the previous value of `capacity()` otherwise. Reallocation invalidates all the references, pointers, and iterators referring to the elements in the sequence. ([vector.capacity], paragraph 3 & paragraph 6)
	`insert()`, `emplace_back()`, `emplace()`, `push_back()`	X	X	Causes reallocation if the new size is greater than the old capacity. If no reallocation happens, all the iterators and references before the insertion point remain valid. ([vector.modifiers], paragraph 1). All iterators and references after the insertion point are invalidated.
	`erase()`, `pop_back()`, `resize()`	X	X	Invalidates iterators and references at or after the point of the erase. ([vector.modifiers], paragraph 3)
	`clear()`	X	X	Destroys all elements in the container. Invalidates all references, pointers, and iterators referring to the elements of the container and may invalidate the past-the-end iterator. ([sequence.reqmts], Table 100)
`std::set`, **`std::multiset`**, **`std::map`**, **`std::multimap`**				
	`erase()`, `clear()`	X	X	Invalidates only iterators and references to the erased elements. ([associative.reqmts], paragraph 9)

Class	Function	Iterators	References/ Pointers	Notes
`std::unordered_set`, **`std::unordered_multiset`**, **`std::unordered_map`**, **`std::unordered_multimap`**				
	`erase()`, `clear()`	X	X	Invalidates only iterators and references to the erased elements. ([unord.req], paragraph 14)
	`insert()`, `emplace()`	X		The `insert` and `emplace` members shall not affect the validity of iterators if $(N+n) < z * B$, where *N* is the number of elements in the container prior to the `insert` operation, *n* is the number of elements inserted, *B* is the container's bucket count, and *z* is the container's maximum load factor. ([unord.req], paragraph 15)
	`rehash()`, `reserve()`	X		Rehashing invalidates iterators, changes ordering between elements, and changes which buckets the elements appear in but does not invalidate pointers or references to elements. ([unord.req], paragraph 9)
`std::valarray`	`resize()`		X	Resizing invalidates all pointers and references to elements in the array. ([valarray.members], paragraph 12)

A `std::basic_string` object is also a container to which this rule applies. For more specific information pertaining to `std::basic_string` containers, see STR52-CPP. Use valid references, pointers, and iterators to reference elements of a basic_string.

5.2.1 Noncompliant Code Example

In this noncompliant code example, `pos` is invalidated after the first call to `insert()`, and subsequent loop iterations have undefined behavior.

```
#include <deque>

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

5.2.2 Compliant Solution (Updated Iterator)

In this compliant solution, `pos` is assigned a valid iterator on each insertion, preventing undefined behavior.

```
#include <deque>

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

5.2.3 Compliant Solution (Generic Algorithm)

This compliant solution replaces the handwritten loop with the generic standard template library algorithm `std::transform()`. The call to `std::transform()` accepts the range of elements to transform, the location to store the transformed values (which, in this case, is a `std::inserter` object to insert them at the beginning of `d`), and the transformation function to apply (which, in this case, is a simple lambda).

```
#include <algorithm>
#include <deque>
#include <iterator>

void f(const double *items, std::size_t count) {
  std::deque<double> d;
  std::transform(items, items + count, std::inserter(d, d.begin()),
                 [](double d) { return d + 41.0; });
}
```

5.2.4 Risk Assessment

Using invalid references, pointers, or iterators to reference elements of a container results in undefined behavior.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
CTR51-CPP	High	Probable	High	**P6**	**L2**

5.2.5 Related Guidelines

SEI CERT C++ Coding Standard	STR52-CPP. Use valid references, pointers, and iterators to reference elements of a basic_string

5.2.6 Bibliography

[ISO/IEC 14882-2014]	Clause 23, “Containers Library” Subclause 24.2.1, “In General”
[Kalev 1999]	*ANSI/ISO C++ Professional Programmer’s Handbook*
[Meyers 2001]	Item 43, “Prefer Algorithm Calls to Handwritten Loops”
[Sutter 2004]	Item 84, “Prefer Algorithm Calls to Handwritten Loops”

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

Copying data into a container that is not large enough to hold that data results in a buffer overflow. To prevent such errors, data copied to the destination container must be restricted on the basis of the destination container's size, or preferably, the destination container must be guaranteed to be large enough to hold the data to be copied.

Vulnerabilities that result from copying data to an undersized buffer can also involve null-terminated strings. Consult STR50-CPP. Guarantee that storage for strings has sufficient space for character data and the null terminator for specific examples of this rule that involve strings.

Copies can be made with the `std::memcpy()` function. However, the `std::memmove()` and `std::memset()` functions can also have the same vulnerabilities because they overwrite a block of memory without checking that the block is valid. Such issues are not limited to C standard library functions; standard template library (STL) generic algorithms, such as `std::copy()`, `std::fill()`, and `std::transform()`, also assume valid output buffer sizes [ISO/IEC 14882-2014].

5.3.1 Noncompliant Code Example

STL containers can be subject to the same vulnerabilities as array data types. The `std::copy()` algorithm provides no inherent bounds checking and can lead to a buffer overflow. In this noncompliant code example, a vector of integers is copied from `src` to `dest` using `std::copy()`. Because `std::copy()` does nothing to expand the `dest` vector, the program will overflow the buffer on copying the first element.

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

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

This hazard applies to any algorithm that takes a destination iterator, expecting to fill it with values. Most of the STL algorithms expect the destination container to have sufficient space to hold the values provided.

5.3.2 Compliant Solution (Sufficient Initial Capacity)

The proper way to use `std::copy()` is to ensure the destination container can hold all the elements being copied to it. This compliant solution enlarges the capacity of the vector prior to the copy operation.

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

5.3.3 Compliant Solution (Per-Element Growth)

An alternative approach is to supply a `std::back_insert_iterator` as the destination argument. This iterator expands the destination container by one element for each element supplied by the algorithm, which guarantees the destination container will become sufficiently large to hold the elements provided.

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

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

5.3.4 Compliant Solution (Assignment)

The simplest solution is to construct `dest` from `src` directly, as in this compliant solution.

```
#include <vector>

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

5.3.5 Noncompliant Code Example

In this noncompliant code example, `std::fill_n()` is used to fill a buffer with 10 instances of the value `0x42`. However, the buffer has not allocated any space for the elements, so this operation results in a buffer overflow.

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

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

5.3.6 Compliant Solution (Sufficient Initial Capacity)

This compliant solution ensures the capacity of the vector is sufficient before attempting to fill the container.

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

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

However, this compliant solution is inefficient. The constructor will default-construct 10 elements of type `int`, which are subsequently replaced by the call to `std::fill_n()`, meaning that each element in the container is initialized twice.

5.3.7 Compliant Solution (Fill Initialization)

This compliant solution initializes `v` to 10 elements whose values are all `0x42`.

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

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

5.3.8 Risk Assessment

Copying data to a buffer that is too small to hold the data results in a buffer overflow. Attackers can exploit this condition to execute arbitrary code.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
CTR52-CPP	High	Likely	Medium	**P18**	**L1**

5.3.9 Related Guidelines

SEI CERT C++ Coding Standard	STR50-CPP. Guarantee that storage for strings has sufficient space for character data and the null terminator
SEI CERT C Coding Standard	ARR38-C. Guarantee that library functions do not form invalid pointers
MITRE CWE	CWE 119, Failure to Constrain Operations within the Bounds of an Allocated Memory Buffer CWE 805, Buffer Access with Incorrect Length Value

5.3.10 Bibliography

[ISO/IEC 14882-2014]	Subclause 25.3, “Mutating Sequence Operations”
[ISO/IEC TR 24772-2013]	Buffer Overflow in Heap [XYB] Buffer Overflow in Stack [XYW] Unchecked Array Indexing [XYZ]
[Meyers 2001]	Item 30, “Make Sure Destination Ranges Are Big Enough”

5.4 CTR53-CPP. Use valid iterator ranges

When iterating over elements of a container, the iterators used must iterate over a valid range. An iterator range is a pair of iterators that refer to the first and past-the-end elements of the range respectively.

A valid iterator range has all of the following characteristics:

- Both iterators refer into the same container.
- The iterator representing the start of the range precedes the iterator representing the end of the range.
- The iterators are not invalidated, in conformance with CTR51-CPP. Use valid references, pointers, and iterators to reference elements of a container.

An empty iterator range (where the two iterators are valid and equivalent) is considered to be valid.

Using a range of two iterators that are invalidated or do not refer into the same container results in undefined behavior.

5.4.1 Noncompliant Code Example

In this noncompliant example, the two iterators that delimit the range point into the same container, but the first iterator does not precede the second. On each iteration of its internal loop, `std::for_each()` compares the first iterator (after incrementing it) with the second for equality; as long as they are not equal, it will continue to increment the first iterator. Incrementing the iterator representing the past-the-end element of the range results in undefined behavior.

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

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

Invalid iterator ranges can also result from comparison functions that return true for equal values. See CTR57-CPP. Provide a valid ordering predicate for more information about comparators.

5.4.2 Compliant Solution

In this compliant solution, the iterator values passed to `std::for_each()` are passed in the proper order.

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

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

5.4.3 Noncompliant Code Example

In this noncompliant code example, iterators from different containers are passed for the same iterator range. Although many STL implementations will compile this code and the program may behave as the developer expects, there is no requirement that an STL implementation treat a default-initialized iterator as a synonym for the iterator returned by `end()`.

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

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

5.4.4 Compliant Solution

In this compliant solution, the proper iterator generated by a call to `end()` is passed.

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

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

5.4.5 Risk Assessment

Using an invalid iterator range is similar to allowing a buffer overflow, which can lead to an attacker running arbitrary code.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
CTR53-CPP	High	Probable	High	**P6**	**L2**

5.4.5.1 Related Vulnerabilities

In *Fun with erase()*, Chris Rohlf discusses the exploit potential of a program that calls `vector::erase()` with invalid iterator ranges [Rohlf 2009].

5.4.6 Related Guidelines

SEI CERT C++ Coding Standard	CTR51-CPP. Use valid references, pointers, and iterators to reference elements of a container CTR57-CPP. Provide a valid ordering predicate

5.4.7 Bibliography

[ISO/IEC 14882-2014]	Clause 24, “Iterators Library” Subclause 25.3, “Mutating Sequence Operations”
[Meyers 2001]	Item 32, “Follow Remove-Like Algorithms with `erase` If You Really Want to Remove Something”

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

When two pointers are subtracted, both must point to elements of the same array object or to one past the last element of the array object; the result is the difference of the subscripts of the two array elements. Similarly, when two iterators are subtracted (including via `std::distance()`), both iterators must refer to the same container object or must be obtained via a call to `end()` (or `cend()`) on the same container object.

If two unrelated iterators (including pointers) are subtracted, the operation results in undefined behavior [ISO/IEC 14882-2014]. Do not subtract two iterators (including pointers) unless both point into the same container or one past the end of the same container.

5.5.1 Noncompliant Code Example

This noncompliant code example attempts to determine whether the pointer `test` is within the range `[r, r + n]`. However, when `test` does not point within the given range, as in this example, the subtraction produces undefined behavior.

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

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

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

5.5.2 Noncompliant Code Example

In this noncompliant code example, the `in_range()` function is implemented using a comparison expression instead of subtraction. The C++ Standard, [expr.rel], paragraph 4 [ISO/IEC 14882-2014], states the following:

> If two operands p and q compare equal, p<=q and p>=q both yield `true` and p<q and p>q both yield `false`. Otherwise, if a pointer p compares greater than a pointer q, p>=q, p>q, q<=p, and q<p all yield `true` and p<=q, p<q, q>=p, and q>p all yield `false`. Otherwise, the result of each of the operators is unspecified.

Thus, comparing two pointers that do not point into the same container or one past the end of the container results in unspecified behavior. Although the following example is an improvement over the previous noncompliant code example, it does not result in portable code and may fail when executed on a segmented memory architecture (such as some antiquated x86 variants). Consequently, it is noncompliant.

```
#include <iostream>

template <typename Ty>
bool in_range(const Ty *test, const Ty *r, size_t n) {
  return test >= r && test < (r + n);
}

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

5.5.3 Noncompliant Code Example

This noncompliant code example is roughly equivalent to the previous example, except that it uses iterators in place of raw pointers. As with the previous example, the `in_range_impl()` function exhibits unspecified behavior when the iterators do not refer into the same container because the operational semantics of `a < b` on a random access iterator are `b - a > 0`, and `>=` is implemented in terms of `<`.

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

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

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

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

5.5.4 Noncompliant Code Example

In this noncompliant code example, `std::less<>` is used in place of the `<` operator. The C++ Standard, [comparisons], paragraph 14 [ISO/IEC 14882-2014], states the following:

> For templates `greater`, `less`, `greater_equal`, and `less_equal`, the specializations for any pointer type yield a total order, even if the built-in operators <, >, <=, >= do not.

Although this approach yields a total ordering, the definition of that total ordering is still unspecified by the implementation. For instance, the following statement could result in the assertion triggering for a given, unrelated pair of pointers, `a` and `b`: `assert(std::less<T *>()(a, b) == std::greater<T *>()(a, b));`. Consequently, this noncompliant code example is still nonportable and, on common implementations of `std::less<>`, may even result in undefined behavior when the `<` operator is invoked.

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

template <typename Ty>
bool in_range(const Ty *test, const Ty *r, size_t n) {
  std::less<const Ty *> less;
  return !less(test, r) && less(test, r + n);
}

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

5.5.5 Compliant Solution

This compliant solution demonstrates a fully portable, but likely inefficient, implementation of `in_range()` that compares `test` against each possible address in the range `[r, n]`. A compliant solution that is both efficient and fully portable is currently unknown.

```
#include <iostream>

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

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

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

5.5.6 Risk Assessment

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
CTR54-CPP	Medium	Probable	Medium	**P8**	**L2**

5.5.7 Related Guidelines

SEI CERT C Coding Standard	ARR36-C. Do not subtract or compare two pointers that do not refer to the same array
MITRE CWE	CWE-469, Use of Pointer Subtraction to Determine Size

5.5.8 Bibliography

[Banahan 2003]	Section 5.3, “Pointers” Section 5.7, “Expressions Involving Pointers”
[ISO/IEC 14882-2014]	Subclause 5.7, “Additive Operators” Subclause 5.9, “Relational Operators” Subclause 20.9.5, “Comparisons”

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

Expressions that have an integral type can be added to or subtracted from a pointer, resulting in a value of the pointer type. If the resulting pointer is not a valid member of the container, or one past the last element of the container, the behavior of the additive operator is undefined. The C++ Standard, [expr.add], paragraph 5 [ISO/IEC 14882-2014], in part, states the following:

> If both the pointer operand and the result point to elements of the same array object, or one past the last element of the array object, the evaluation shall not produce an overflow; otherwise, the behavior is undefined.

Because iterators are a generalization of pointers, the same constraints apply to additive operators with random access iterators. Specifically, the C++ Standard, [iterator.requirements.general], paragraph 5, states the following:

> Just as a regular pointer to an array guarantees that there is a pointer value pointing past the last element of the array, so for any iterator type there is an iterator value that points past the last element of a corresponding sequence. These values are called *past-the-end* values. Values of an iterator `i` for which the expression `*i` is defined are called *dereferenceable*. The library never assumes that past-the-end values are dereferenceable.

Do not allow an expression of integral type to add to or subtract from a pointer or random access iterator when the resulting value would overflow the bounds of the container.

5.6.1 Noncompliant Code Example (`std::vector`)

In this noncompliant code example, a random access iterator from a `std::vector` is used in an additive expression, but the resulting value could be outside the bounds of the container rather than a past-the-end value.

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

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

5.6.2 Compliant Solution (`std::vector`)

This compliant solution assumes that the programmer's intention was to process up to 20 items in the container. Instead of assuming all containers will have 20 or more elements, the size of the container is used to determine the upper bound on the addition.

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

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

5.6.3 Risk Assessment

If adding or subtracting an integer to a pointer results in a reference to an element outside the array or one past the last element of the array object, the behavior is undefined but frequently leads to a buffer overflow or buffer underrun, which can often be exploited to run arbitrary code. Iterators and standard template library containers exhibit the same behavior and caveats as pointers and arrays.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
CTR55-CPP	High	Likely	Medium	**P18**	**L1**

5.6.4 Related Guidelines

SEI CERT C Coding Standard	ARR30-C. Do not form or use out-of-bounds pointers or array subscripts
MITRE CWE	CWE 129, Unchecked Array Indexing

5.6.5 Bibliography

[Banahan 2003]	Section 5.3, "Pointers" Section 5.7, "Expressions Involving Pointers"
[ISO/IEC 14882-2014]	Subclause 5.7, "Additive Operators" Subclause 24.2.1, "In General"
[VU#162289]	

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

The definition of *pointer arithmetic* from the C++ Standard, [expr.add], paragraph 7 [ISO/IEC 14882-2014], states the following:

> For addition or subtraction, if the expressions `P` or `Q` have type "pointer to *cv* `T`", where `T` is different from the cv-unqualified array element type, the behavior is undefined. [*Note:* In particular, a pointer to a base class cannot be used for pointer arithmetic when the array contains objects of a derived class type. *—end note*]

Pointer arithmetic does not account for polymorphic object sizes, and attempting to perform pointer arithmetic on a polymorphic object value results in undefined behavior.

The C++ Standard, [expr.sub], paragraph 1 [ISO/IEC 14882-2014], defines array subscripting as being identical to pointer arithmetic. Specifically, it states the following:

> The expression `E1[E2]` is identical (by definition) to `*((E1)+(E2))`.

Do not use pointer arithmetic, including array subscripting, on polymorphic objects.

The following code examples assume the following static variables and class definitions.

```
int globI;
double globD;

struct S {
  int i;

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

struct T : S {
  double d;

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

5.7.1 Noncompliant Code Example (Pointer Arithmetic)

In this noncompliant code example, `f()` accepts an array of `S` objects as its first parameter. However, `main()` passes an array of `T` objects as the first argument to `f()`, which results in undefined behavior due to the pointer arithmetic used within the `for` loop.

```
#include <iostream>

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

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

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

5.7.2 Noncompliant Code Example (Array Subscripting)

In this noncompliant code example, the `for` loop uses array subscripting. Since array subscripts are computed using pointer arithmetic, this code also results in undefined behavior.

```
#include <iostream>

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

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

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

5.7.3 Compliant Solution (Array)

Instead of having an array of objects, an array of pointers solves the problem of the objects being of different sizes, as in this compliant solution.

```
#include <iostream>

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

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

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

The elements in the arrays are no longer polymorphic objects (instead, they are pointers to polymorphic objects), and so there is no undefined behavior with the pointer arithmetic.

5.7.4 Compliant Solution (`std::vector`)

Another approach is to use a standard template library (STL) container instead of an array and have `f()` accept iterators as parameters, as in this compliant solution. However, because STL containers require homogeneous elements, pointers are still required within the container.

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

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

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

5.7.5 Risk Assessment

Using arrays polymorphically can result in memory corruption, which could lead to an attacker being able to execute arbitrary code.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
CTR56-CPP	High	Likely	High	**P9**	**L2**

5.7.6 Related Guidelines

SEI CERT C Coding Standard	ARR39-C. Do not add or subtract a scaled integer to a pointer

5.7.7 Bibliography

[ISO/IEC 14882-2014]	Subclause 5.7, “Additive Operators” Subclause 5.2.1, “Subscripting”
[Lockheed Martin 05]	AV Rule 96, “Arrays shall not be treated polymorphically”
[Meyers 96]	Item 3, “Never Treat Arrays Polymorphically”
[Stroustrup 2006]	“What’s Wrong with Arrays?”
[Sutter 2004]	Item 100, “Don’t Treat Arrays Polymorphically”

5.8 CTR57-CPP. Provide a valid ordering predicate

Associative containers place a strict weak ordering requirement on their key comparison predicates [ISO/IEC 14882-2014]. A strict weak ordering has the following properties:

- for all `x`: `x < x == false` (irreflexivity)
- for all `x`, `y`: if `x < y` then `!(y < x)` (asymmetry)
- for all `x`, `y`, `z`: if `x < y && y < z then x < z` (transitivity)

Providing an invalid ordering predicate for an associative container (e.g., sets, maps, multisets, and multimaps), or as a comparison criterion with the sorting algorithms, can result in erratic behavior or infinite loops [Meyers 2001]. When an ordering predicate is required for an associative container or a generic standard template library algorithm, the predicate must meet the requirements for inducing a strict weak ordering.

5.8.1 Noncompliant Code Example

In this noncompliant code example, the `std::set` object is created with a comparator that does not adhere to the strict weak ordering requirement. Specifically, it fails to return false for equivalent values. As a result, the behavior of iterating over the results from `std::set::equal_range` results in unspecified behavior.

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

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

5.8.2 Compliant Solution

This compliant solution uses the default comparator with `std::set` instead of providing an invalid one.

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

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

5.8.3 Noncompliant Code Example

In this noncompliant code example, the objects stored in the `std::set` have an overloaded `operator<` implementation, allowing the objects to be compared with `std::less`. However, the comparison operation does not provide a strict weak ordering. Specifically, two sets, `x` and `y`, whose `i` values are both 1, but have differing `j` values can result in a situation where `comp(x, y)` and `comp(y, x)` are both false, failing the asymmetry requirements.

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

class S {
  int i, j;

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

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

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

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

5.8.4 Compliant Solution

This compliant solution uses `std::tie()` to properly implement the strict weak ordering `operator<` predicate.

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

class S {
  int i, j;

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

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

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

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

5.8.5 Risk Assessment

Using an invalid ordering rule can lead to erratic behavior or infinite loops.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
CTR57-CPP	Low	Probable	High	**P2**	**L3**

5.8.6 Related Guidelines

SEI CERT Oracle Coding Standard for Java	MET10-J. Follow the general contract when implementing the compareTo() method

5.8.7 Bibliography

[ISO/IEC 14882-2014]	Subclause 23.2.4, “Associative Containers”
[Meyers 2001]	Item 21, “Always Have Comparison Functions Return False for Equal Values”
[Sutter 2004]	Item 83, “Use a Checked STL Implementation”

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

The C++ standard library implements numerous common algorithms that accept a predicate function object. The C++ Standard, [algorithms.general], paragraph 10 [ISO/IEC 14882-2014], states the following:

> [*Note:* Unless otherwise specified, algorithms that take function objects as arguments are permitted to copy those function objects freely. Programmers for whom object identity is important should consider using a wrapper class that points to a noncopied implementation object such as `reference_wrapper<T>`, or some equivalent solution. — *end note*]

Because it is implementation-defined whether an algorithm copies a predicate function object, any such object must either

- implement a function call operator that does not mutate state related to the function object's identity, such as nonstatic data members or captured values, or
- wrap the predicate function object in a `std::reference_wrapper<T>` (or an equivalent solution).

Marking the function call operator as `const` is beneficial, but insufficient, because data members with the `mutable` storage class specifier may still be modified within a `const` member function.

5.9.1 Noncompliant Code Example (Functor)

This noncompliant code example attempts to remove the third item in a container using a predicate that returns `true` only on its third invocation.

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

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

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

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

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

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

However, `std::remove_if()` is permitted to construct and use extra copies of its predicate function. Any such extra copies may result in unexpected output.

5.9.1.1 Implementation Details

This program produces the following results using GCC 4.8.1 with libstdc++.

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

This result arises because `std::remove_if` makes two copies of the predicate before invoking it. The first copy is used to determine the location of the first element in the sequence for which the predicate returns `true`. The subsequent copy is used for removing other elements in the sequence. This results in the third element (`2`) and sixth element (`5`) being removed; two distinct predicate functions are used.

5.9.2 Noncompliant Code Example (Lambda)

Similar to the functor noncompliant code example, this noncompliant code example attempts to remove the third item in a container using a predicate lambda function that returns `true` only on its third invocation. As with the functor, this lambda carries local state information, which it attempts to mutate within its function call operator.

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

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

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

  int timesCalled = 0;
  v.erase(std::remove_if(
              v.begin(),
              v.end(),
              [timesCalled](const int &) mutable {
                  return ++timesCalled == 3;
              }),
          v.end());
  print_container(v.begin(), v.end());
}
```

5.9.3 Compliant Solution (`std::reference_wrapper`)

This compliant solution wraps the predicate in a `std::reference_wrapper<T>` object, ensuring that copies of the wrapper object all refer to the same underlying predicate object.

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

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

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

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

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

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

The above compliant solution demonstrates using a reference wrapper over a functor object but can similarly be used with a stateful lambda. The code produces the expected results, where only the third element is removed.

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

5.9.4 Compliant Solution (Iterator Arithmetic)

Removing a specific element of a container does not require a predicate function but can instead simply use `std::vector::erase()`, as in this compliant solution.

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

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

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

5.9.5 Risk Assessment

Using a predicate function object that contains state can produce unexpected values.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
CTR58-CPP	Low	Likely	High	**P3**	**L3**

5.9.6 Bibliography

[ISO/IEC 14882-2014]	Subclause 25.1, “General”
[Meyers 2001]	Item 39, “Make Predicates Pure Functions”

6 Characters and Strings (STR)

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

Copying data to a buffer that is not large enough to hold that data results in a buffer overflow. Buffer overflows occur frequently when manipulating strings [Seacord 2013]. To prevent such errors, either limit copies through truncation or, preferably, ensure that the destination is of sufficient size to hold the data to be copied. C-style strings require a null character to indicate the end of the string, while the C++ `std::basic_string` template requires no such character.

6.1.1 Noncompliant Code Example

Because the input is unbounded, the following code could lead to a buffer overflow.

```
#include <iostream>

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

6.1.2 Noncompliant Code Example

To solve this problem, it may be tempting to use the `std::ios_base::width()` method, but there still is a trap, as shown in this noncompliant code example.

```
#include <iostream>

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

In this example, the first read will not overflow, but could fill `bufOne` with a truncated string. Furthermore, the second read still could overflow `bufTwo`. The C++ Standard, [istream.extractors], paragraphs 7–9 [ISO/IEC 14882-2014], describes the behavior of `operator>>(basic_istream &, charT *)` and, in part, states the following:

> `operator>>` then stores a null byte (`charT()`) in the next position, which may be the first position if no characters were extracted. `operator>>` then calls `width(0)`.

Consequently, it is necessary to call `width()` prior to each `operator>>` call passing a bounded array. However, this does not account for the input being truncated, which may lead to information loss or a possible vulnerability.

6.1.3 Compliant Solution

The best solution for ensuring that data is not truncated and for guarding against buffer overflows is to use `std::string` instead of a bounded array, as in this compliant solution.

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

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

6.1.4 Noncompliant Code Example

In this noncompliant example, the unformatted input function `std::basic_istream<T>::read()` is used to read an unformatted character array of 32 characters from the given file. However, the `read()` function does not guarantee that the string will be null terminated, so the subsequent call of the `std::string` constructor results in undefined behavior if the character array does not contain a null terminator.

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

6.1.5 Compliant Solution

This compliant solution assumes that the input from the file is at most 32 characters. Instead of inserting a null terminator, it constructs the `std::string` object based on the number of characters read from the input stream. If the size of the input is uncertain, it is better to use `std::basic_istream<T>::readsome()` or a formatted input function, depending on need.

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

6.1.6 Risk Assessment

Copying string data to a buffer that is too small to hold that data results in a buffer overflow. Attackers can exploit this condition to execute arbitrary code with the permissions of the vulnerable process.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
STR50-CPP	High	Likely	Medium	**P18**	**L1**

6.1.7 Related Guidelines

SEI CERT C Coding Standard	STR31-C. Guarantee that storage for strings has sufficient space for character data and the null terminator

6.1.8 Bibliography

[ISO/IEC 14882-2014]	Subclause 27.7.2.2.3, “`basic_istream::operator>>`” Subclause 27.7.2.3, “Unformatted Input Functions”
[Seacord 2013]	Chapter 2, “Strings”

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

The `std::basic_string` type uses the *traits* design pattern to handle implementation details of the various string types, resulting in a series of string-like classes with a common, underlying implementation. Specifically, the `std::basic_string` class is paired with `std::char_traits` to create the `std::string`, `std::wstring`, `std::u16string`, and `std::u32string` classes. The `std::char_traits` class is explicitly specialized to provide policy-based implementation details to the `std::basic_string` type. One such implementation detail is the `std::char_traits::length()` function, which is frequently used to determine the number of characters in a null-terminated string. According to the C++ Standard, [char.traits.require], Table 62 [ISO/IEC 14882-2014], passing a null pointer to this function is undefined behavior because it would result in dereferencing a null pointer.

The following `std::basic_string` member functions result in a call to `std::char_traits::length()`:

- `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 *)`

The following `std::basic_string` nonmember functions result in a call to `std::char_traits::length()`:

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

Do not call any of the preceding functions with a null pointer as the `const charT *` argument.

This rule is a specific instance of EXP34-C. Do not dereference null pointers.

6.2.1.1 Implementation Details

Some standard library vendors, such as libstdc++, throw a `std::logic_error` when a null pointer is used in the above function calls, though not when calling `std::char_traits::length()`. However, `std::logic_error` is not a requirement of the C++ Standard, and some vendors (e.g., libc++ and the Microsoft Visual Studio STL) do not implement this behavior. For portability, you should not rely on this behavior.

6.2.2 Noncompliant Code Example

In this noncompliant code example, a `std::string` object is created from the results of a call to `std::getenv()`. However, because `std::getenv()` returns a null pointer on failure, this code can lead to undefined behavior when the environment variable does not exist (or some other error occurs).

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

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

6.2.3 Compliant Solution

In this compliant solution, the results from the call to `std::getenv()` are checked for null before the `std::string` object is constructed.

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

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

6.2.4 Risk Assessment

Dereferencing a null pointer is undefined behavior, typically abnormal program termination. In some situations, however, dereferencing a null pointer can lead to the execution of arbitrary code [Jack 2007, van Sprundel 2006]. The indicated severity is for this more severe case; on platforms where it is not possible to exploit a null pointer dereference to execute arbitrary code, the actual severity is low.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
STR51-CPP	High	Likely	Medium	**P18**	**L1**

6.2.5 Related Guidelines

SEI CERT C Coding Standard	EXP34-C. Do not dereference null pointers

6.2.6 Bibliography

[ISO/IEC 9899:2011]	Subclause 7.20.3, “Memory Management Functions”
[ISO/IEC 14882-2014]	Subclause 21.2.1, “Character Trait Requirements”
[Jack 2007]	
[van Sprundel 2006]	

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

Since `std::basic_string` is a container of characters, this rule is a specific instance of CTR51-CPP. Use valid references, pointers, and iterators to reference elements of a container. As a container, it supports iterators just like other containers in the Standard Template Library. However, the `std::basic_string` template class has unusual invalidation semantics. The C++ Standard, [string.require], paragraph 5 [ISO/IEC 14882-2014], states the following:

> References, pointers, and iterators referring to the elements of a `basic_string` sequence may be invalidated by the following uses of that `basic_string` object:
>
> As an argument to any standard library function taking a reference to non-const `basic_string` as an argument.
>
> Calling non-const member functions, except `operator[]`, `at`, `front`, `back`, `begin`, `rbegin`, `end`, and `rend`.

Examples of standard library functions taking a reference to non-`const` `std::basic_string` are `std::swap()`, `::operator>>(basic_istream &, string &)`, and `std::getline()`.

Do not use an invalidated reference, pointer, or iterator because doing so results in undefined behavior.

6.3.1 Noncompliant Code Example

This noncompliant code example copies `input` into a `std::string`, replacing semicolon (`;`) characters with spaces. This example is noncompliant because the iterator `loc` is invalidated after the first call to `insert()`. The behavior of subsequent calls to `insert()` is undefined.

```
#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 : ' ');
  }
}
```

6.3.2 Compliant Solution (`std::string::insert()`)

In this compliant solution, the value of the iterator `loc` is updated as a result of each call to `insert()` so that the invalidated iterator is never accessed. The updated iterator is then incremented at the end of the loop.

```
#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 : ' ');
  }
}
```

6.3.3 Compliant Solution (`std::replace()`)

This compliant solution uses a standard algorithm to perform the replacement. When possible, using a generic algorithm is preferable to inventing your own solution.

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

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

6.3.4 Noncompliant Code Example

In this noncompliant code example, `data` is invalidated after the call to `replace()`, and so its use in `g()` is undefined behavior.

```
#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);
}
```

6.3.5 Compliant Solution

In this compliant solution, the pointer to `exampleString`'s internal buffer is not generated until after the modification from `replace()` has completed.

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

extern void g(const char *);

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

6.3.6 Risk Assessment

Using an invalid reference, pointer, or iterator to a string object could allow an attacker to run arbitrary code.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
STR52-CPP	High	Probable	High	**P6**	**L2**

6.3.7 Related Guidelines

SEI CERT C++ Coding Standard	CTR51-CPP. Use valid references, pointers, and iterators to reference elements of a container

6.3.8 Bibliography

[ISO/IEC 14882-2014]	Subclause 21.4.1, “`basic_string` General Requirements”
[Meyers 2001]	Item 43, “Prefer Algorithm Calls to Hand-written Loops”

6.4 STR53-CPP. Range check element access

The `std::string` index operators `const_reference operator[](size_type) const` and `reference operator[](size_type)` return the character stored at the specified position, `pos`. When `pos >= size()`, a reference to an object of type `charT` with value `charT()` is returned. The index operators are unchecked (no exceptions are thrown for range errors), and attempting to modify the resulting out-of-range object results in undefined behavior.

Similarly, the `std::string::back()` and `std::string::front()` functions are unchecked as they are defined to call through to the appropriate `operator[]()` without throwing.

Do not pass an out-of-range value as an argument to `std::string::operator[]()`. Similarly, do not call `std::string::back()` or `std::string::front()` on an empty string. This rule is a specific instance of CTR50-CPP. Guarantee that container indices and iterators are within the valid range.

6.4.1 Noncompliant Code Example

In this noncompliant code example, the value returned by the call to `get_index()` may be greater than the number of elements stored in the string, resulting in undefined behavior.

```
#include <string>

extern std::size_t get_index();

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

6.4.2 Compliant Solution (`try/catch`)

This compliant solution uses the `std::basic_string::at()` function, which behaves in a similar fashion to the index `operator[]` but throws a `std::out_of_range` exception if `pos >= size()`.

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

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

6.4.3 Compliant Solution (Range Check)

This compliant solution checks that the value returned by `get_index()` is within a valid range before calling `operator[]()`.

```
#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
  }
}
```

6.4.4 Noncompliant Code Example

This noncompliant code example attempts to replace the initial character in the string with a capitalized equivalent. However, if the given string is empty, the behavior is undefined.

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

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

6.4.5 Compliant Solution

In this compliant solution, the call to `std::string::front()` is made only if the string is not empty.

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

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

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

6.4.6 Risk Assessment

Unchecked element access can lead to out-of-bound reads and writes and write-anywhere exploits. These exploits can, in turn, lead to the execution of arbitrary code with the permissions of the vulnerable process.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
STR53-CPP	High	Unlikely	Medium	**P6**	**L2**

6.4.7 Related Guidelines

SEI CERT C++ Coding Standard	CTR50-CPP. Guarantee that container indices and iterators are within the valid range

6.4.8 Bibliography

[ISO/IEC 14882-2014]	Subclause 21.4.5, “`basic_string` Element Access”
[Seacord 2013]	Chapter 2, “Strings”

7 Memory Management (MEM)

7.1 MEM50-CPP. Do not access freed memory

Evaluating a pointer—including dereferencing the pointer, using it as an operand of an arithmetic operation, type casting it, and using it as the right-hand side of an assignment—into memory that has been deallocated by a memory management function is undefined behavior. Pointers to memory that has been deallocated are called *dangling pointers*. Accessing a dangling pointer can result in exploitable vulnerabilities.

It is at the memory manager's discretion when to reallocate or recycle the freed memory. When memory is freed, all pointers into it become invalid, and its contents might either be returned to the operating system, making the freed space inaccessible, or remain intact and accessible. As a result, the data at the freed location can appear to be valid but change unexpectedly. Consequently, memory must not be written to or read from once it is freed.

7.1.1 Noncompliant Code Example (`new` and `delete`)

In this noncompliant code example, `s` is dereferenced after it has been deallocated. If this access results in a write-after-free, the vulnerability can be exploited to run arbitrary code with the permissions of the vulnerable process. Typically, dynamic memory allocations and deallocations are far removed, making it difficult to recognize and diagnose such problems.

```
#include <new>

struct S {
  void f();
};

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

The function `g()` is marked `noexcept(false)` to comply with MEM52-CPP. Detect and handle memory allocation errors.

7.1.2 Compliant Solution (`new` and `delete`)

In this compliant solution, the dynamically allocated memory is not deallocated until it is no longer required.

```
#include <new>

struct S {
  void f();
};

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

7.1.3 Compliant Solution (Automatic Storage Duration)

When possible, use automatic storage duration instead of dynamic storage duration. Since `s` is not required to live beyond the scope of `g()`, this compliant solution uses automatic storage duration to limit the lifetime of `s` to the scope of `g()`.

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

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

7.1.4 Noncompliant Code Example (`std::unique_ptr`)

In the following noncompliant code example, the dynamically allocated memory managed by the `buff` object is accessed after it has been implicitly deallocated by the object's destructor.

```
#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;
}
```

This code always creates a null-terminated byte string, despite its use of `strncpy()`, because it leaves the final `char` in the buffer set to 0.

7.1.5 Compliant Solution (`std::unique_ptr`)

In this compliant solution, the lifetime of the `buff` object extends past the point at which the memory managed by the object is accessed.

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

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

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

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

7.1.6 Compliant Solution

In this compliant solution, a variable with automatic storage duration of type `std::string` is used in place of the `std::unique_ptr<char[]>`, which reduces the complexity and improves the security of the solution.

```
#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;
}
```

7.1.7 Noncompliant Code Example (`std::string::c_str()`)

In this noncompliant code example, `std::string::c_str()` is being called on a temporary `std::string` object. The resulting pointer will point to released memory once the `std::string` object is destroyed at the end of the assignment expression, resulting in undefined behavior when accessing elements of that pointer.

```
#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 */
}
```

7.1.8 Compliant solution (`std::string::c_str()`)

In this compliant solution, a local copy of the string returned by `str_func()` is made to ensure that string `str` will be valid when the call to `display_string()` is made.

```
#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 */
}
```

7.1.9 Noncompliant Code Example

In this noncompliant code example, an attempt is made to allocate zero bytes of memory through a call to `operator new()`. If this request succeeds, `operator new()` is required to return a non-null pointer value. However, according to the C++ Standard, [basic.stc.dynamic.allocation], paragraph 2 [ISO/IEC 14882-2014], attempting to dereference memory through such a pointer results in undefined behavior.

```
#include <new>

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

7.1.10 Compliant Solution

The compliant solution depends on programmer intent. If the programmer intends to allocate a single `unsigned char` object, the compliant solution is to use `new` instead of a direct call to `operator new()`, as this compliant solution demonstrates.

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

7.1.11 Compliant Solution

If the programmer intends to allocate zero bytes of memory (perhaps to obtain a unique pointer value that cannot be reused by any other pointer in the program until it is properly released), then instead of attempting to dereference the resulting pointer, the recommended solution is to declare `ptr` as a `void *`, which cannot be dereferenced by a conforming implementation.

```
#include <new>

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

7.1.12 Risk Assessment

Reading previously dynamically allocated memory after it has been deallocated can lead to abnormal program termination and denial-of-service attacks. Writing memory that has been deallocated can lead to the execution of arbitrary code with the permissions of the vulnerable process.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
MEM50-CPP	High	Likely	Medium	**P18**	**L1**

7.1.12.1 Related Vulnerabilities

VU#623332 describes a double-free vulnerability in the MIT Kerberos 5 function krb5_recvauth() [VU#623332].

7.1.13 Related Guidelines

SEI CERT C++ Coding Standard	EXP54-CPP. Do not access an object outside of its lifetime MEM52-CPP. Detect and handle memory allocation errors
SEI CERT C Coding Standard	MEM30-C. Do not access freed memory
MITRE CWE	CWE-415, Double Free CWE-416, Use After Free

7.1.14 Bibliography

[ISO/IEC 14882-2014]	Subclause 3.7.4.1, “Allocation Functions” Subclause 3.7.4.2, “Deallocation Functions”
[Seacord 2013b]	Chapter 4, “Dynamic Memory Management”

7.2 MEM51-CPP. Properly deallocate dynamically allocated resources

The C programming language provides several ways to allocate memory, such as `std::malloc()`, `std::calloc()`, and `std::realloc()`, which can be used by a C++ program. However, the C programming language defines only a single way to free the allocated memory: `std::free()`. See MEM31-C. Free dynamically allocated memory when no longer needed and MEM34-C. Only free memory allocated dynamically for rules specifically regarding C allocation and deallocation requirements.

The C++ programming language adds additional ways to allocate memory, such as the operators `new`, `new[]`, and placement `new`, and allocator objects. Unlike C, C++ provides multiple ways to free dynamically allocated memory, such as the operators `delete`, `delete[]()`, and deallocation functions on allocator objects.

Do not call a deallocation function on anything other than `nullptr`, or a pointer returned by the corresponding allocation function described by the following.

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

Passing a pointer value to an inappropriate deallocation function can result in undefined behavior.

The C++ Standard, [expr.delete], paragraph 2 [ISO/IEC 14882-2014], in part, states the following:

> In the first alternative (*delete object*), the value of the operand of `delete` may be a null pointer value, a pointer to a non-array object created by a previous *new-expression*, or a pointer to a subobject (1.8) representing a base class of such an object (Clause 10). If not, the behavior is undefined. In the second alternative (*delete array*), the value of the operand of `delete` may be a null pointer value or a pointer value that resulted from a previous array *new-expression*. If not, the behavior is undefined.

Deallocating a pointer that is not allocated dynamically (including non-dynamic pointers returned from calls to placement `new()`) is undefined behavior because the pointer value was not obtained by an allocation function. Deallocating a pointer that has already been passed to a deallocation function is undefined behavior because the pointer value no longer points to memory that has been dynamically allocated.

When an operator such as `new` is called, it results in a call to an overloadable operator of the same name, such as `operator new()`. These overloadable functions can be called directly but carry the same restrictions as their operator counterparts. That is, calling `operator delete()` and passing a pointer parameter has the same constraints as calling the `delete` operator on that pointer. Further, the overloads are subject to scope resolution, so it is possible (but not permissible) to call a class-specific operator to allocate an object but a global operator to deallocate the object.

See MEM53-CPP. Explicitly construct and destruct objects when manually managing object lifetime for information on lifetime management of objects when using memory management functions other than the `new` and `delete` operators.

7.2.1 Noncompliant Code Example (placement `new()`)

In this noncompliant code example, the local variable `s1` is passed as the expression to the placement `new` operator. The resulting pointer of that call is then passed to `::operator delete()`, resulting in undefined behavior due to `::operator delete()` attempting to free memory that was not returned by `::operator new()`.

```
#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;
}
```

7.2.2 Compliant Solution (placement `new()`)

This compliant solution removes the call to `::operator delete()`, allowing `s1` to be destroyed as a result of its normal object lifetime termination.

```
#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;

  // ...
}
```

7.2.3 Noncompliant Code Example (Uninitialized `delete`)

In this noncompliant code example, two allocations are attempted within the same `try` block, and if either fails, the `catch` handler attempts to free resources that have been allocated. However, because the pointer variables have not been initialized to a known value, a failure to allocate memory for `i1` may result in passing `::operator delete()` a value (in `i2`) that was not previously returned by a call to `::operator new()`, resulting in undefined behavior.

```
#include <new>

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

7.2.4 Compliant Solution (Uninitialized `delete`)

This compliant solution initializes both pointer values to `nullptr`, which is a valid value to pass to `::operator delete()`.

```
#include <new>

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

7.2.5 Noncompliant Code Example (Double-Free)

Once a pointer is passed to the proper deallocation function, that pointer value is invalidated. Passing the pointer to a deallocation function a second time when the pointer value has not been returned by a subsequent call to an allocation function results in an attempt to free memory that has not been allocated dynamically. The underlying data structures that manage the heap can become corrupted in a way that can introduce security vulnerabilities into a program. These types of issues are called *double-free vulnerabilities*. In practice, double-free vulnerabilities can be exploited to execute arbitrary code.

In this noncompliant code example, the class `C` is given ownership of a `P *`, which is subsequently deleted by the class destructor. The C++ Standard, [class.copy], paragraph 7 [ISO/IEC 14882-2014], states the following:

> If the class definition does not explicitly declare a copy constructor, one is declared implicitly. If the class definition declares a move constructor or move assignment operator, the implicitly declared copy constructor is defined as deleted; otherwise, it is defined as defaulted (8.4). The latter case is deprecated if the class has a user-declared copy assignment operator or a user-declared destructor.

Despite the presence of a user-declared destructor, `C` will have an implicitly defaulted copy constructor defined for it, and this defaulted copy constructor will copy the pointer value stored in `p`, resulting in a double-free: the first free happens when `g()` exits and the second free happens when `h()` exits.

```
struct P {};

class C {
  P *p;

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

  void f() {}
};

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

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

7.2.6 Compliant Solution (Double-Free)

In this compliant solution, the copy constructor and copy assignment operator for `C` are explicitly deleted. This deletion would result in an ill-formed program with the definition of `g()` from the preceding noncompliant code example due to use of the deleted copy constructor. Consequently, `g()` was modified to accept its parameter by reference, removing the double-free.

```
struct P {};

class C {
  P *p;

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

  void operator=(const C&) = delete;

  void f() {}
};

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

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

7.2.7 Noncompliant Code Example (array `new[]`)

In the following noncompliant code example, an array is allocated with array `new[]` but is deallocated with a scalar `delete` call instead of an array `delete[]` call, resulting in undefined behavior.

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

7.2.8 Compliant Solution (array `new[]`)

In the compliant solution, the code is fixed by replacing the call to `delete` with a call to `delete []` to adhere to the correct pairing of memory allocation and deallocation functions.

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

7.2.9 Noncompliant Code Example (`malloc()`)

In this noncompliant code example, the call to `malloc()` is mixed with a call to `delete`.

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

This code does not violate MEM53-CPP. Explicitly construct and destruct objects when manually managing object lifetime because it complies with the MEM53-CPP-EX1 exception.

7.2.9.1 Implementation Details

Some implementations of `::operator new()` result in calling `std::malloc()`. On such implementations, the `::operator delete()` function is required to call `std::free()` to deallocate the pointer, and the noncompliant code example would behave in a well-defined manner. However, this is an implementation detail and should not be relied on—implementations are under no obligation to use underlying C memory management functions to implement C++ memory management operators.

7.2.10 Compliant Solution (`malloc()`)

In this compliant solution, the pointer allocated by `std::malloc()` is deallocated by a call to `std::free()` instead of `delete`.

```
#include <cstdlib>

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

7.2.11 Noncompliant Code Example (new)

In this noncompliant code example, `std::free()` is called to deallocate memory that was allocated by `new`. A common side effect of the undefined behavior caused by using the incorrect deallocation function is that destructors will not be called for the object being deallocated by `std::free()`.

```
#include <cstdlib>

struct S {
  ~S();
};

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

Additionally, this code violates MEM53-CPP. Explicitly construct and destruct objects when manually managing object lifetime.

7.2.12 Compliant Solution (new)

In this compliant solution, the pointer allocated by `new` is deallocated by calling `delete` instead of `std::free()`.

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

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

7.2.13 Noncompliant Code Example (Class new)

In this noncompliant code example, the global `new` operator is overridden by a class-specific implementation of `operator new()`. When `new` is called, the class-specific override is selected, so `S::operator new()` is called. However, because the object is destroyed with a scoped `::delete` operator, the global `operator delete()` function is called instead of the class-specific implementation `S::operator delete()`, resulting in undefined behavior.

```
#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;
}
```

7.2.14 Compliant Solution (class new)

In this compliant solution, the scoped `::delete` call is replaced by a nonscoped `delete` call, resulting in `S::operator delete()` being called.

```
#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;
}
```

7.2.15 Noncompliant Code Example (`std::unique_ptr`)

In this noncompliant code example, a `std::unique_ptr` is declared to hold a pointer to an object, but is direct-initialized with an array of objects. When the `std::unique_ptr` is destroyed, its default deleter calls `delete` instead of `delete[]`, resulting in undefined behavior.

```
#include <memory>

struct S {};

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

7.2.16 Compliant Solution (`std::unique_ptr`)

In this compliant solution, the `std::unique_ptr` is declared to hold an array of objects instead of a pointer to an object. Additionally, `std::make_unique()` is used to initialize the smart pointer.

```
#include <memory>

struct S {};

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

Use of `std::make_unique()` instead of direct initialization will emit a diagnostic if the resulting `std::unique_ptr` is not of the correct type. Had it been used in the noncompliant code example, the result would have been an ill-formed program instead of undefined behavior. It is best to use `std::make_unique()` instead of manual initialization by other means.

7.2.17 Noncompliant Code Example (`std::shared_ptr`)

In this noncompliant code example, a `std::shared_ptr` is declared to hold a pointer to an object, but is direct-initialized with an array of objects. As with `std::unique_ptr`, when the `std::shared_ptr` is destroyed, its default deleter calls `delete` instead of `delete[]`, resulting in undefined behavior.

```
#include <memory>

struct S {};

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

7.2.18 Compliant Solution (`std::shared_ptr`)

Unlike the compliant solution for `std::unique_ptr`, where `std::make_unique()` is called to create a unique pointer to an array, it is ill-formed to call `std::make_shared()` with an array type. Instead, this compliant solution manually specifies a custom deleter for the shared pointer type, ensuring that the underlying array is properly deleted.

```
#include <memory>

struct S {};

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

7.2.19 Risk Assessment

Passing a pointer value to a deallocation function that was not previously obtained by the matching allocation function results in undefined behavior, which can lead to exploitable vulnerabilities.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
MEM51-CPP	High	Likely	Medium	**P18**	**L1**

7.2.20 Related Guidelines

SEI CERT C++ Coding Standard	MEM53-CPP. Explicitly construct and destruct objects when manually managing object lifetime
SEI CERT C Coding Standard	MEM31-C. Free dynamically allocated memory when no longer needed MEM34-C. Only free memory allocated dynamically
MITRE CWE	CWE 590, Free of Memory Not on the Heap CWE 415, Double Free CWE 404, Improper Resource Shutdown or Release CWE 762, Mismatched Memory Management Routines

7.2.21 Bibliography

[Dowd 2007]	"Attacking `delete` and `delete []` in C++"
[Henricson 1997]	Rule 8.1, "`delete` should only be used with `new`" Rule 8.2, "`delete []` should only be used with `new []`"
[ISO/IEC 14882-2014]	Subclause 5.3.5, "Delete" Subclause 12.8, "Copying and Moving Class Objects" Subclause 18.6.1, "Storage Allocation and Deallocation" Subclause 20.7.11, "Temporary Buffers"
[Meyers 2005]	Item 16, "Use the Same Form in Corresponding Uses of `new` and `delete`"
[Seacord 2013]	Chapter 4, "Dynamic Memory Management"
[Viega 2005]	"Doubly Freeing Memory"

7.3 MEM52-CPP. Detect and handle memory allocation errors

The default memory allocation operator, `::operator new(std::size_t)`, throws a `std::bad_alloc` exception if the allocation fails. Therefore, you need not check whether calling `::operator new(std::size_t)` results in `nullptr`. The nonthrowing form, `::operator new(std::size_t, const std::nothrow_t &)`, does not throw an exception if the allocation fails but instead returns `nullptr`. The same behaviors apply for the `operator new[]` versions of both allocation functions. Additionally, the default allocator object (`std::allocator`) uses `::operator new(std::size_t)` to perform allocations and should be treated similarly.

```
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
```

Furthermore, `operator new[]` can throw an error of type `std::bad_array_new_length`, a subclass of `std::bad_alloc`, if the `size` argument passed to `new` is negative or excessively large.

When using the nonthrowing form, it is imperative to check that the return value is not `nullptr` before accessing the resulting pointer. When using either form, be sure to comply with ERR50-CPP. Do not abruptly terminate the program.

7.3.1 Noncompliant Code Example

In this noncompliant code example, an array of `int` is created using `::operator new[](std::size_t)` and the results of the allocation are not checked. The function is marked as `noexcept`, so the caller assumes this function does not throw any exceptions. Because `::operator new[](std::size_t)` can throw an exception if the allocation fails, it could lead to abnormal termination of the program.

```
#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;
}
```

7.3.2 Compliant Solution (`std::nothrow`)

When using `std::nothrow`, the `new` operator returns either a null pointer or a pointer to the allocated space. Always test the returned pointer to ensure it is not `nullptr` before referencing the pointer. This compliant solution handles the error condition appropriately when the returned pointer is `nullptr`.

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

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

7.3.3 Compliant Solution (`std::bad_alloc`)

Alternatively, you can use `::operator new[]` without `std::nothrow` and instead catch a `std::bad_alloc` exception if sufficient memory cannot be allocated.

```
#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;
}
```

7.3.4 Compliant Solution (noexcept(false))

If the design of the function is such that the caller is expected to handle exceptional situations, it is permissible to mark the function explicitly as one that may throw, as in this compliant solution. Marking the function is not strictly required, as any function without a `noexcept` specifier is presumed to allow throwing.

```
#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;
}
```

7.3.5 Noncompliant Code Example

In this noncompliant code example, two memory allocations are performed within the same expression. Because the memory allocations are passed as arguments to a function call, an exception thrown as a result of one of the calls to `new` could result in a memory leak.

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

void g(A *, B *);
void f() {
  g(new A, new B);
}
```

Consider the situation in which `A` is allocated and constructed first, and then `B` is allocated and throws an exception. Wrapping the call to `g()` in a `try`/`catch` block is insufficient because it would be impossible to free the memory allocated for `A`.

This noncompliant code example also violates EXP50-CPP. Do not depend on the order of evaluation for side effects, because the order in which the arguments to `g()` are evaluated is unspecified.

7.3.6 Compliant Solution (`std::unique_ptr`)

In this compliant solution, a `std::unique_ptr` is used to manage the resources for the A and B objects with RAII. In the situation described by the noncompliant code example, B throwing an exception would still result in the destruction and deallocation of the A object when then `std::unique_ptr<A>` was destroyed.

```
#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>());
}
```

7.3.7 Compliant Solution (References)

When possible, the more resilient compliant solution is to remove the memory allocation entirely and pass the objects by reference instead.

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

void g(A &a, B &b);
void f() {
  A a;
  B b;
  g(a, b);
}
```

7.3.8 Risk Assessment

Failing to detect allocation failures can lead to abnormal program termination and denial-of-service attacks.

If the vulnerable program references memory offset from the return value, an attacker can exploit the program to read or write arbitrary memory. This vulnerability has been used to execute arbitrary code [VU#159523].

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
MEM52-CPP	High	Likely	Medium	**P18**	**L1**

7.3.8.1 Related Vulnerabilities

The vulnerability in Adobe Flash [VU#159523] arises because Flash neglects to check the return value from `calloc()`. Even though `calloc()` returns `NULL`, Flash does not attempt to read or write to the return value. Instead, it attempts to write to an offset from the return value. Dereferencing `NULL` usually results in a program crash, but dereferencing an offset from `NULL` allows an exploit to succeed without crashing the program.

7.3.9 Related Guidelines

SEI CERT C Coding Standard	ERR33-C. Detect and handle standard library errors
MITRE CWE	CWE 252, Unchecked Return Value CWE 391, Unchecked Error Condition CWE 476, NULL Pointer Dereference CWE 690, Unchecked Return Value to NULL Pointer Dereference CWE 703, Improper Check or Handling of Exceptional Conditions CWE 754, Improper Check for Unusual or Exceptional Conditions

7.3.10 Bibliography

[ISO/IEC 9899:2011]	Subclause 7.20.3, “Memory Management Functions”
[ISO/IEC 14882-2014]	Subclause 18.6.1.1, “Single-Object Forms” Subclause 18.6.1.2, “Array Forms” Subclause 20.7.9.1, “Allocator Members”
[Meyers 1996]	Item 7, “Be Prepared for Out-of-Memory Conditions”
[Seacord 2013]	Chapter 4, “Dynamic Memory Management”

7.4 MEM53-CPP. Explicitly construct and destruct objects when manually managing object lifetime

The creation of dynamically allocated objects in C++ happens in two stages. The first stage is responsible for allocating sufficient memory to store the object, and the second stage is responsible for initializing the newly allocated chunk of memory, depending on the type of the object being created.

Similarly, the destruction of dynamically allocated objects in C++ happens in two stages. The first stage is responsible for finalizing the object, depending on the type, and the second stage is responsible for deallocating the memory used by the object. The C++ Standard, [basic.life], paragraph 1 [ISO/IEC 14882-2014], states the following:

> The *lifetime* of an object is a runtime property of the object. An object is said to have non-trivial initialization if it is of a class or aggregate type and it or one of its members is initialized by a constructor other than a trivial default constructor. [*Note:* initialization by a trivial copy/move constructor is non-trivial initialization. — *end note*] The lifetime of an object of type T begins when:
>
> - storage with the proper alignment and size for type T is obtained, and
> - if the object has non-trivial initialization, its initialization is complete.
>
> The lifetime of an object of type T ends when:
>
> - if T is a class type with a non-trivial destructor, the destructor call starts, or
> - the storage which the object occupies is reused or released.

For a dynamically allocated object, these two stages are typically handled automatically by using the `new` and `delete` operators. The expression `new T` for a type `T` results in a call to `operator new()` to allocate sufficient memory for `T`. If memory is successfully allocated, the default constructor for `T` is called. The result of the expression is a pointer `P` to the object of type `T`. When that pointer is passed in the expression `delete P`, it results in a call to the destructor for `T`. After the destructor completes, a call is made to `operator delete()` to deallocate the memory.

When a program creates a dynamically allocated object by means other than the `new` operator, it is said to be *manually managing* the lifetime of that object. This situation arises when using other allocation schemes to obtain storage for the dynamically allocated object, such as using an allocator object or `malloc()`. For example, a custom container class may allocate a slab of memory in a `reserve()` function in which subsequent objects will be stored. See MEM51-CPP. Properly deallocate dynamically allocated resources for further information on dynamic memory management.

When manually managing the lifetime of an object, the constructor must be called to initiate the lifetime of the object. Similarly, the destructor must be called to terminate the lifetime of the object. Use of an object outside of its lifetime is undefined behavior. An object can be constructed either by calling the constructor explicitly using the placement `new` operator or by calling the `construct()` function of an allocator object. An object can be destroyed either by calling the destructor explicitly or by calling the `destroy()` function of an allocator object.

7.4.1 Noncompliant Code Example

In this noncompliant code example, a class with nontrivial initialization (due to the presence of a user-provided constructor) is created with a call to `std::malloc()`. However, the constructor for the object is never called, resulting in undefined behavior when the class is later accessed by calling `s->f()`.

```
#include <cstdlib>

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

void g() {
  S *s = static_cast<S *>(std::malloc(sizeof(S)));
  s->f();
  std::free(s);
}
```

7.4.2 Compliant Solution

In this compliant solution, the constructor and destructor are both explicitly called. Further, to reduce the possibility of the object being used outside of its lifetime, the underlying storage is a separate variable from the live object.

```
#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);
}
```

7.4.3 Noncompliant Code Example

In this noncompliant code example, a custom container class uses an allocator object to obtain storage for arbitrary element types. While the `copy_elements()` function is presumed to call copy constructors for elements being moved into the newly allocated storage, this example fails to explicitly call the default constructor for any additional elements being reserved. If such an element is accessed through the `operator[]()` function, it results in undefined behavior, depending on the type `T`.

```
#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]; }
};
```

7.4.4 Compliant Solution

In this compliant solution, all elements are properly initialized by explicitly calling copy or default constructors for `T`.

```
#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];
  }
};
```

7.4.5 Exceptions

MEM53-CPP-EX1: If the object is trivially constructable, it need not have its constructor explicitly called to initiate the object’s lifetime. If the object is trivially destructible, it need not have its destructor explicitly called to terminate the object’s lifetime. These properties can be tested by calling `std::is_trivially_constructible()` and `std::is_trivially_destructible()` from `<type_traits>`. For instance, integral types such as `int` and `long long` do not require an explicit constructor or destructor call.

7.4.6 Risk Assessment

Failing to properly construct or destroy an object leaves its internal state inconsistent, which can result in undefined behavior and accidental information exposure.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
MEM53-CPP	High	Likely	Medium	**P18**	**L1**

7.4.7 Related Guidelines

SEI CERT C++ Coding Standard	MEM51-CPP. Properly deallocate dynamically allocated resources

7.4.8 Bibliography

[ISO/IEC 14882-2014]	Subclause 3.8, “Object Lifetime” Clause 9, “Classes”

7.5 MEM54-CPP. Provide placement new with properly aligned pointers to sufficient storage capacity

When invoked by a `new` expression for a given type, the default global non-placement forms of C++ `operator new` attempt to allocate sufficient storage for an object of the type and, if successful, return a pointer with alignment suitable for any object with a fundamental alignment requirement. However, the default placement `new` operator simply returns the given pointer back to the caller without guaranteeing that there is sufficient space in which to construct the object or ensuring that the pointer meets the proper alignment requirements. The C++ Standard, [expr.new], paragraph 16 [ISO/IEC 14882-2014], nonnormatively states the following:

> [Note: when the allocation function returns a value other than null, it must be a pointer to a block of storage in which space for the object has been reserved. The block of storage is assumed to be appropriately aligned and of the requested size. The address of the created object will not necessarily be the same as that of the block if the object is an array. —end note]

(This note is a reminder of the general requirements specified by the C++ Standard, [basic.stc.dynamic.allocation], paragraph 1, which apply to placement `new` operators by virtue of [basic.stc.dynamic], paragraph 3.)

In addition, the standard provides the following example later in the same section:

> `new(2, f) T[5]` results in a call `of operator new[](sizeof(T) * 5 + y, 2, f).`
>
> Here, `...` and `y` are non-negative unspecified values representing array allocation overhead; the result of the new-expression will be offset by this amount from the value returned by `operator new[]`. This overhead may be applied in all array *new-expressions*, including those referencing the library function `operator new[](std::size_t, void*)` and other placement allocation functions. The amount of overhead may vary from one invocation of new to another.

Do not pass a pointer that is not suitably aligned for the object being constructed to placement `new`. Doing so results in an object being constructed at a misaligned location, which results in undefined behavior. Do not pass a pointer that has insufficient storage capacity for the object being constructed, including the overhead required for arrays. Doing so may result in initialization of memory outside of the bounds of the object being constructed, which results in undefined behavior.

7.5.1 Noncompliant Code Example

In this noncompliant code example, a pointer to a `short` is passed to placement `new`, which is attempting to initialize a `long`. On architectures where `sizeof(short) < sizeof(long)`, this results in undefined behavior. This example, and subsequent ones, all assume the pointer returned by placement `new` will not be used after the lifetime of its underlying storage has ended. For instance, the pointer will not be stored in a `static` global variable and dereferenced after the call to `f()` has ended. This assumption is in conformance with MEM50-CPP. Do not access freed memory.

```
#include <new>

void f() {
  short s;
  long *lp = ::new (&s) long;
}
```

7.5.2 Noncompliant Code Example

This noncompliant code example ensures that the `long` is constructed into a buffer of sufficient size. However, it does not ensure that the alignment requirements are met for the pointer passed into placement `new`. To make this example clearer, an additional local variable `c` has also been declared.

```
#include <new>

void f() {
  char c; // Used elsewhere in the function
  unsigned char buffer[sizeof(long)];
  long *lp = ::new (buffer) long;

  // ...
}
```

7.5.3 Compliant Solution (`alignas`)

In this compliant solution, the `alignas` declaration specifier is used to ensure the buffer is appropriately aligned for a `long`.

```
#include <new>

void f() {
  char c; // Used elsewhere in the function
  alignas(long) unsigned char buffer[sizeof(long)];
  long *lp = ::new (buffer) long;

  // ...
}
```

7.5.4 Compliant Solution (`std::aligned_storage`)

This compliant solution ensures that the `long` is constructed into a buffer of sufficient size and with suitable alignment.

```
#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;

  // ...
}
```

7.5.5 Noncompliant Code Example (Failure to Account for Array Overhead)

This noncompliant code example attempts to allocate sufficient storage of the appropriate alignment for the array of objects of S. However, it fails to account for the overhead an implementation may add to the amount of storage for array objects. The overhead (commonly referred to as a cookie) is necessary to store the number of elements in the array so that the array delete expression or the exception unwinding mechanism can invoke the type’s destructor on each successfully constructed element of the array. While some implementations are able to avoid allocating space for the cookie in some situations, assuming they do in all cases is unsafe.

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

7.5.6 Compliant Solution (Clang/GCC)

The amount of overhead required by array new expressions is unspecified but ideally would be documented by quality implementations. The following compliant solution is specifically for the Clang and GNU GCC compilers, which guarantee that the overhead for dynamic array allocations is a single value of type `size_t`. To verify that the assumption is, in fact, safe, the compliant solution also overloads the placement `new[]` operator to accept the buffer size as a third argument and verifies that it is at least as large as the total amount of storage required.

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

Porting this compliant solution to other implementations requires adding similar conditional definitions of the overhead constant, depending on the constraints of the platform.

7.5.7 Risk Assessment

Passing improperly aligned pointers or pointers to insufficient storage to placement `new` expressions can result in undefined behavior, including buffer overflow and abnormal termination.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
MEM54-CPP	High	Likely	Medium	**P18**	**L1**

7.5.8 Related Guidelines

SEI CERT C++ Coding Standard	MEM50-CPP. Do not access freed memory

7.5.9 Bibliography

[ISO/IEC 14882-2014]	Subclause 3.7.4, "Dynamic Storage Duration" Subclause 5.3.4, "New"

7.6 MEM55-CPP. Honor replacement dynamic storage management requirements

Dynamic memory allocation and deallocation functions can be globally replaced by custom implementations, as specified by [replacement.functions], paragraph 2, of the C++ Standard [ISO/IEC 14882-2014]. For instance, a user may profile the dynamic memory usage of an application and decide that the default allocator is not optimal for their usage pattern, and a different allocation strategy may be a marked improvement. However, the C++ Standard, [res.on.functions], paragraph 1, states the following:

> In certain cases (replacement functions, handler functions, operations on types used to instantiate standard library template components), the C++ standard library depends on components supplied by a C++ program. If these components do not meet their requirements, the Standard places no requirements on the implementation.

Paragraph 2 further, in part, states the following:

> In particular, the effects are undefined in the following cases:
>
> - for replacement functions, if the installed replacement function does not implement the semantics of the applicable *Required behavior:* paragraph.

A replacement for any of the dynamic memory allocation or deallocation functions must meet the semantic requirements specified by the appropriate *Required behavior:* clause of the replaced function.

7.6.1 Noncompliant Code Example

In this noncompliant code example, the global `operator new(std::size_t)` function is replaced by a custom implementation. However, the custom implementation fails to honor the behavior required by the function it replaces, as per the C++ Standard, [new.delete.single], paragraph 3. Specifically, if the custom allocator fails to allocate the requested amount of memory, the replacement function returns a null pointer instead of throwing an exception of type `std::bad_alloc`. By returning a null pointer instead of throwing, functions relying on the required behavior of `operator new(std::size_t)` to throw on memory allocations may instead attempt to dereference a null pointer. See EXP34-C. Do not dereference null pointers for more information.

```
#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
```

The declarations of the replacement `operator delete()` functions indicate that this noncompliant code example still complies with DCL54-CPP. Overload allocation and deallocation functions as a pair in the same scope.

7.6.2 Compliant Solution

This compliant solution implements the required behavior for the replaced global allocator function by properly throwing a `std::bad_alloc` exception when the allocation fails.

```
#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
```

7.6.3 Risk Assessment

Failing to meet the stated requirements for a replaceable dynamic storage function leads to undefined behavior. The severity of risk depends heavily on the caller of the allocation functions, but in some situations, dereferencing a null pointer can lead to the execution of arbitrary code [Jack 2007, van Sprundel 2006]. The indicated severity is for this more severe case.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
MEM55-CPP	High	Likely	Medium	**P18**	**L1**

7.6.4 Related Guidelines

SEI CERT C++ Coding Standard	DCL54-CPP. Overload allocation and deallocation functions as a pair in the same scope OOP56-CPP. Honor replacement handler requirements
SEI CERT C Coding Standard	EXP34-C. Do not dereference null pointers

7.6.5 Bibliography

[ISO/IEC 14882-2014]	Subclause 17.6.4.8, “Other Functions” Subclause 18.6.1, “Storage Allocation and Deallocation”
[Jack 2007]	
[van Sprundel 2006]	

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

Smart pointers such as `std::unique_ptr` and `std::shared_ptr` encode pointer ownership semantics as part of the type system. They wrap a pointer value, provide pointer-like semantics through `operator *()` and `operator->()` member functions, and control the lifetime of the pointer they manage. When a smart pointer is constructed from a pointer value, that value is said to be *owned* by the smart pointer.

Calling `std::unique_ptr::release()` will relinquish ownership of the managed pointer value. Destruction of, move assignment of, or calling `std::unique_ptr::reset()` on a `std::unique_ptr` object will also relinquish ownership of the managed pointer value, but results in destruction of the managed pointer value. If a call to `std::shared_ptr::unique()` returns true, then destruction of or calling `std::shared_ptr::reset()` on that `std::shared_ptr` object will relinquish ownership of the managed pointer value but results in destruction of the managed pointer value.

Some smart pointers, such as `std::shared_ptr`, allow multiple smart pointer objects to manage the same underlying pointer value. In such cases, the initial smart pointer object owns the pointer value, and subsequent smart pointer objects are related to the original smart pointer. Two smart pointers are *related* when the initial smart pointer is used in the initialization of the subsequent smart pointer objects. For instance, copying a `std::shared_ptr` object to another `std::shared_ptr` object via copy assignment creates a relationship between the two smart pointers, whereas creating a `std::shared_ptr` object from the managed pointer value of another `std::shared_ptr` object does not.

Do not create an unrelated smart pointer object with a pointer value that is owned by another smart pointer object. This includes resetting a smart pointer's managed pointer to an already-owned pointer value, such as by calling `reset()`.

7.7.1 Noncompliant Code Example

In this noncompliant code example, two unrelated smart pointers are constructed from the same underlying pointer value. When the local, automatic variable `p2` is destroyed, it deletes the pointer value it manages. Then, when the local, automatic variable `p1` is destroyed, it deletes the same pointer value, resulting in a double-free vulnerability.

```
#include <memory>

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

7.7.2 Compliant Solution

In this compliant solution, the `std::shared_ptr` objects are related to one another through copy construction. When the local, automatic variable `p2` is destroyed, the use count for the shared pointer value is decremented but still nonzero. Then, when the local, automatic variable `p1` is destroyed, the use count for the shared pointer value is decremented to zero, and the managed pointer is destroyed. This compliant solution also calls `std::make_shared`() instead of allocating a raw pointer and storing its value in a local variable.

```
#include <memory>

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

7.7.3 Noncompliant Code Example

In this noncompliant code example, the `poly` pointer value owned by a `std::shared_ptr` object is cast to the `D *` pointer type with `dynamic_cast` in an attempt to obtain a `std::shared_ptr` of the polymorphic derived type. However, this eventually results in undefined behavior as the same pointer is thereby stored in two different `std::shared_ptr` objects. When `g()` exits, the pointer stored in `derived` is freed by the default deleter. Any further use of `poly` results in accessing freed memory. When `f()` exits, the same pointer stored in `poly` is destroyed, resulting in a double-free vulnerability.

```
#include <memory>

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

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

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

7.7.4 Compliant Solution

In this compliant solution, the `dynamic_cast` is replaced with a call to `std::dynamic_pointer_cast()`, which returns a `std::shared_ptr` of the polymorphic type with the valid shared pointer value. When `g()` exits, the reference count to the underlying pointer is decremented by the destruction of `derived`, but because of the reference held by `poly` (within `f()`), the stored pointer value is still valid after `g()` returns.

```
#include <memory>

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

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

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

7.7.5 Noncompliant Code Example

In this noncompliant code example, a `std::shared_ptr` of type `S` is constructed and stored in `s1`. Later, `S::g()` is called to get another shared pointer to the pointer value managed by `s1`. However, the smart pointer returned by `S::g()` is not related to the smart pointer stored in `s1`. When `s2` is destroyed, it will free the pointer managed by `s1`, causing a double-free vulnerability when `s1` is destroyed.

```
#include <memory>

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

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

7.7.6 Compliant Solution

The compliant solution is to use `std::enable_shared_from_this::shared_from_this()` to get a shared pointer from `S` that is related to an existing `std::shared_ptr` object. A common implementation strategy is for the `std::shared_ptr` constructors to detect the presence of a pointer that inherits from `std::enable_shared_from_this`, and automatically update the internal bookkeeping required for `std::enable_shared_from_this::shared_from_this()` to work. Note that `std::enable_shared_from_this::shared_from_this()` requires an existing `std::shared_ptr` instance that manages the pointer value pointed to by `this`. Failure to meet this requirement results in undefined behavior, as it would result in a smart pointer attempting to manage the lifetime of an object that itself does not have lifetime management semantics.

```
#include <memory>

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

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

7.7.7 Risk Assessment

Passing a pointer value to a deallocation function that was not previously obtained by the matching allocation function results in undefined behavior, which can lead to exploitable vulnerabilities.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
MEM56-CPP	High	Likely	Medium	**P18**	**L1**

7.7.8 Related Guidelines

SEI CERT C++ Coding Standard	MEM50-CPP. Do not access freed memory MEM51-CPP. Properly deallocate dynamically allocated resources
MITRE CWE	CWE-415, Double Free CWE-416, Use After Free CWE 762, Mismatched Memory Management Routines

7.7.9 Bibliography

[ISO/IEC 14882-2014]	Subclause 20.8, “Smart Pointers”

7.8 MEM57-CPP. Avoid using default operator new for over-aligned types

The non-placement `new` expression is specified to invoke an allocation function to allocate storage for an object of the specified type. When successful, the allocation function, in turn, is required to return a pointer to storage with alignment suitable for any object with a fundamental alignment requirement. Although the global `operator new`, the default allocation function invoked by the new expression, is specified by the C++ standard [ISO/IEC 14882-2014] to allocate sufficient storage suitably aligned to represent any object of the specified size, since the expected alignment isn't part of the function's interface, the most a program can safely assume is that the storage allocated by the default `operator new` defined by the implementation is aligned for an object with a fundamental alignment. In particular, it is unsafe to use the storage for an object of a type with a stricter alignment requirement—an *over-aligned type*.

Furthermore, the array form of the non-placement `new` expression may increase the amount of storage it attempts to obtain by invoking the corresponding allocation function by an unspecified amount. This amount, referred to as overhead in the C++ standard, is commonly known as a *cookie*. The cookie is used to store the number of elements in the array so that the array delete expression or the exception unwinding mechanism can invoke the type's destructor on each successfully constructed element of the array. While the specific conditions under which the cookie is required by the array new expression aren't described in the C++ standard, they may be outlined in other specifications such as the application binary interface (ABI) document for the target environment. For example, the Itanium C++ ABI describes the rules for computing the size of a cookie, its location, and achieving the correct alignment of the array elements. When these rules require that a cookie be created, it is possible to obtain a suitably aligned array of elements of an overaligned type [CodeSourcery 2016a]. However, the rules are complex and the Itanium C++ ABI isn't universally applicable.

Avoid relying on the default `operator new` to obtain storage for objects of over-aligned types. Doing so may result in an object being constructed at a misaligned location, which has undefined behavior and can result in abnormal termination when the object is accessed, even on architectures otherwise known to tolerate misaligned accesses.

7.8.1 Noncompliant Code Example

In the following noncompliant code example, the new expression is used to invoke the default `operator new` to obtain storage in which to then construct an object of the user-defined type `Vector` with alignment that exceeds the fundamental alignment of most implementations (typically 16 bytes). Objects of such over-aligned types are typically required by SIMD (single instruction, multiple data) vectorization instructions, which can trap when passed unsuitably aligned arguments.

```
struct alignas(32) Vector {
  char elems[32];
};

Vector *f() {
  Vector *pv = new Vector;
  return pv;
}
```

7.8.2 Compliant Solution (`aligned_alloc`)

In this compliant solution, an overloaded `operator new` function is defined to obtain appropriately aligned storage by calling the C11 function `aligned_alloc()`. Programs that make use of the array form of the new expression must define the corresponding member array `operator new[]` and `operator delete[]`. The `aligned_alloc()` function is not part of the C++ 98, C++ 11, or C++ 14 standards but may be provided by implementations of such standards as an extension. Programs targeting C++ implementations that do not provide the C11 `aligned_alloc()` function must define the member `operator new` to adjust the alignment of the storage obtained by the allocation function of their choice.

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

struct alignas(32) Vector {
  char elems[32];
  static void *operator new(size_t nbytes) {
    if (void *p = std::aligned_alloc(alignof(Vector), nbytes)) {
      return p;
    }
    throw std::bad_alloc();
  }
  static void operator delete(void *p) {
    free(p);
  }
};

Vector *f() {
  Vector *pv = new Vector;
  return pv;
}
```

7.8.3 Risk Assessment

Using improperly aligned pointers results in undefined behavior, typically leading to abnormal termination.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
MEM57-CPP	Medium	Unlikely	Low	**P6**	**L2**

7.8.4 Related Guidelines

SEI CERT C++ Coding Standard	MEM54-CPP. Provide placement new with properly aligned pointers to sufficient storage capacity

7.8.5 Bibliography

[ISO/IEC 14882-2014]	Subclause 3.7.4, “Dynamic Storage Duration” Subclause 5.3.4, “New” Subclause 18.6.1, “Storage Allocation and Deallocation”
[CodeSourcery 2016a]	Itanium C++ ABI, version 1.86
[INCITS 2012]	Dynamic memory allocation for over-aligned data, WG14 proposal

8 Input Output (FIO)

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

The C++ Standard, [filebuf], paragraph 2 [ISO/IEC 14882-2014], states the following:

> The restrictions on reading and writing a sequence controlled by an object of class `basic_filebuf<charT, traits>` are the same as for reading and writing with the Standard C library `FILE`s.

The C Standard, subclause 7.19.5.3, paragraph 6 [ISO/IEC 9899:1999], places the following restrictions on `FILE` objects opened for both reading and writing:

> When a file is opened with update mode . . ., both input and output may be performed on the associated stream. However, output shall not be directly followed by input without an intervening call to the `fflush` function or to a file positioning function (`fseek`, `fsetpos`, or `rewind`), and input shall not be directly followed by output without an intervening call to a file positioning function, unless the input operation encounters end-of-file.

Consequently, the following scenarios can result in undefined behavior:

- Receiving input from a stream directly following an output to that stream without an intervening call to `std::basic_filebuf<T>::seekoff()` if the file is not at end-of-file
- Outputting to a stream after receiving input from that stream without a call to `std::basic_filebuf<T>::seekoff()` if the file is not at end-of-file

No other `std::basic_filebuf<T>` function guarantees behavior as if a call were made to a standard C library file-positioning function, or `std::fflush()`.

Calling `std::basic_ostream<T>::seekp()` or `std::basic_istream<T>::seekg()` eventually results in a call to `std::basic_filebuf<T>::seekoff()` for file stream positioning. Given that `std::basic_iostream<T>` inherits from both `std:: basic_ostream<T>` and `std::basic_istream<T>`, and `std::fstream` inherits from `std::basic_iostream`, either function is acceptable to call to ensure the file buffer is in a valid state before the subsequent I/O operation.

8.1.1 Noncompliant Code Example

This noncompliant code example appends data to the end of a file and then reads from the same file. However, because there is no intervening positioning call between the formatted output and input calls, the behavior is undefined.

```
#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;
}
```

8.1.2 Compliant Solution

In this compliant solution, the `std::basic_istream<T>::seekg()` function is called between the output and input, eliminating the undefined behavior.

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

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

  file << "Output some data";

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

8.1.3 Risk Assessment

Alternately inputting and outputting from a stream without an intervening flush or positioning call is undefined behavior.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
FIO50-CPP	Low	Likely	Medium	**P6**	**L2**

8.1.4 Related Guidelines

This rule supplements FIO39-C. Do not alternately input and output from a stream without an intervening flush or positioning call.

8.1.5 Bibliography

[ISO/IEC 9899:1999]	Subclause 7.19.5.3, “The `fopen` Function”
[ISO/IEC 14882-2014]	Clause 27, “Input/Output Library”

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

A call to the `std::basic_filebuf<T>::open()` function must be matched with a call to `std::basic_filebuf<T>::close()` before the lifetime of the last pointer that stores the return value of the call has ended or before normal program termination, whichever occurs first.

Note that `std::basic_ifstream<T>`, `std::basic_ofstream<T>`, and `std::basic_fstream<T>` all maintain an internal reference to a `std::basic_filebuf<T>` object on which `open()` and `close()` are called as needed. Properly managing an object of one of these types (by not leaking the object) is sufficient to ensure compliance with this rule. Often, the best solution is to use the stream object by value semantics instead of via dynamic memory allocation, ensuring compliance with MEM51-CPP. Properly deallocate dynamically allocated resources. However, that is still insufficient for situations in which destructors are not automatically called.

8.2.1 Noncompliant Code Example

In this noncompliant code example, a `std::fstream` object `file` is constructed. The constructor for `std::fstream` calls `std::basic_filebuf<T>::open()`, and the default `std::terminate_handler` called by `std::terminate()` is `std::abort()`, which does not call destructors. Consequently, the underlying `std::basic_filebuf<T>` object maintained by the object is not properly closed.

```
#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();
}
```

This noncompliant code example and the subsequent compliant solutions are assumed to eventually call `std::terminate()` in accordance with the ERR50-CPP-EX1 exception described in ERR50-CPP. Do not abruptly terminate the program. Indicating the nature of the problem to the operator is elided for brevity.

8.2.2 Compliant Solution

In this compliant solution, `std::fstream::close()` is called before `std::terminate()` is called, ensuring that the file resources are properly closed.

```
#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();
}
```

8.2.3 Compliant Solution

In this compliant solution, the stream is implicitly closed through RAII before `std::terminate()` is called, ensuring that the file resources are properly closed.

```
#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();
}
```

8.2.4 Risk Assessment

Failing to properly close files may allow an attacker to exhaust system resources and can increase the risk that data written into in-memory file buffers will not be flushed in the event of abnormal program termination.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
FIO51-CPP	Medium	Unlikely	Medium	**P4**	**L3**

8.2.5 Related Guidelines

This rule supplements FIO42-C. Close files when they are no longer needed.

SEI CERT C++ Coding Standard	MEM51-CPP. Properly deallocate dynamically allocated resources

8.2.6 Bibliography

[ISO/IEC 14882-2014]	Subclause 27.9.1, “File Streams”

9 Exceptions and Error Handling (ERR)

9.1 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()`.

9.1.1 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
  }
  // ...
}
```

9.1.2 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
  }
  // ...
}
```

9.1.3 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()`.

9.1.4 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**

9.1.5 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

9.1.6 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)

9.2 ERR51-CPP. Handle all exceptions

When an exception is thrown, control is transferred to the nearest handler with a type that matches the type of the exception thrown. If no matching handler is directly found within the handlers for a try block in which the exception is thrown, the search for a matching handler continues to dynamically search for handlers in the surrounding try blocks of the same thread. The C++ Standard, [except.handle], paragraph 9 [ISO/IEC 14882-2014], states the following:

> If no matching handler is found, the function `std::terminate()` is called; whether or not the stack is unwound before this call to `std::terminate()` is implementation-defined.

The default terminate handler called by `std::terminate()` calls `std::abort()`, which abnormally terminates the process. When `std::abort()` is called, or if the implementation does not unwind the stack prior to calling `std::terminate()`, destructors for objects may not be called and external resources can be left in an indeterminate state. Abnormal process termination is the typical vector for denial-of-service attacks. For more information on implicitly calling `std::terminate()`, see ERR50-CPP. Do not abruptly terminate the program.

All exceptions thrown by an application must be caught by a matching exception handler. Even if the exception cannot be gracefully recovered from, using the matching exception handler ensures that the stack will be properly unwound and provides an opportunity to gracefully manage external resources before terminating the process.

As per **ERR50-CPP-EX1**, a program that encounters an unrecoverable exception may explicitly catch the exception and terminate, but it may not allow the exception to remain uncaught. One possible solution to comply with this rule, as well as with ERR50-CPP, is for the `main()` function to catch all exceptions. While this does not generally allow the application to recover from the exception gracefully, it does allow the application to terminate in a controlled fashion.

9.2.1 Noncompliant Code Example

In this noncompliant code example, neither `f()` nor `main()` catch exceptions thrown by `throwing_func()`. Because no matching handler can be found for the exception thrown, `std::terminate()` is called.

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

void f() {
  throwing_func();
}

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

9.2.2 Compliant Solution

In this compliant solution, the main entry point handles all exceptions, which ensures that the stack is unwound up to the `main()` function and allows for graceful management of external resources.

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

void f() {
  throwing_func();
}

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

9.2.3 Noncompliant Code Example

In this noncompliant code example, the thread entry point function `thread_start()` does not catch exceptions thrown by `throwing_func()`. If the initial thread function exits because an exception is thrown, `std::terminate()` is called.

```
#include <thread>

void throwing_func() noexcept(false);

void thread_start() {
  throwing_func();
}

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

9.2.4 Compliant Solution

In this compliant solution, the `thread_start()` handles all exceptions and does not rethrow, allowing the thread to terminate normally.

```
#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();
}
```

9.2.5 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
ERR51-CPP	Low	Probable	Medium	**P4**	**L3**

9.2.6 Related Guidelines

This rule is a subset of ERR50-CPP. Do not abruptly terminate the program.

MITRE CWE	CWE-754, Improper Check for Unusual or Exceptional Conditions

9.2.7 Bibliography

[ISO/IEC 14882-2014]	Subclause 15.1, “Throwing an Exception” Subclause 15.3, “Handling an Exception” Subclause 15.5.1, “The `std::terminate()` Function”
[MISRA 2008]	Rule 15-3-2 (Advisory) Rule 15-3-4 (Required)

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

The C standard library facilities `setjmp()` and `longjmp()` can be used to simulate throwing and catching exceptions. However, these facilities bypass automatic resource management and can result in undefined behavior, commonly including resource leaks and denial-of-service attacks.

The C++ Standard, [support.runtime], paragraph 4 [ISO/IEC 14882-2014], states the following:

> The function signature `longjmp(jmp_buf jbuf, int val)` has more restricted behavior in this International Standard. A `setjmp`/`longjmp` call pair has undefined behavior if replacing the `setjmp` and `longjmp` by `catch` and `throw` would invoke any non-trivial destructors for any automatic objects.

Do not call `setjmp()` or `longjmp()`; their usage can be replaced by more standard idioms such as `throw` expressions and `catch` statements.

9.3.1 Noncompliant Code Example

If a `throw` expression would cause a nontrivial destructor to be invoked, then calling `longjmp()` in the same context will result in undefined behavior. In the following noncompliant code example, the call to `longjmp()` occurs in a context with a local `Counter` object. Since this object's destructor is nontrivial, undefined behavior results.

```
#include <csetjmp>
#include <iostream>
static jmp_buf env;

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

int Counter::instances = 0;
void f() {
  Counter c;
  std::cout << "f(): Instances: "
            << Counter::instances << std::endl;
  std::longjmp(env, 1);
}

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

9.3.1.1 Implementation Details

The above code produces the following results when compiled with Clang 3.8 for Linux, demonstrating that the program, on this platform, fails to destroy the local `Counter` instance when the execution of `f()` is terminated. This is permissible as the behavior is undefined.

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

9.3.2 Compliant Solution

This compliant solution replaces the calls to `setjmp()` and `longjmp()` with a `throw` expression and a `catch` statement.

```
#include <iostream>

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

int Counter::instances = 0;

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

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

This solution produces the following output.

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

9.3.3 Risk Assessment

Using `setjmp()` and `longjmp()` could lead to a denial-of-service attack due to resources not being properly destroyed.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
ERR52-CPP	Low	Probable	Medium	**P4**	**L3**

9.3.4 Bibliography

[Henricson 1997]	Rule 13.3, Do not use `setjmp()` and `longjmp()`
[ISO/IEC 14882-2014]	Subclause 18.10, “Other Runtime Support”

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

When an exception is caught by a *function-try-block* handler in a constructor, any fully constructed base classes and class members of the object are destroyed prior to entering the handler [ISO/IEC 14882-2014]. Similarly, when an exception is caught by a *function-try-block* handler in a destructor, all base classes and nonvariant class members of the objects are destroyed prior to entering the handler. Because of this behavior, the C++ Standard, [except.handle], paragraph 10, states the following:

> Referring to any non-static member or base class of an object in the handler for a *function-try-block* of a constructor or destructor for that object results in undefined behavior.

Do not reference base classes or class data members in a constructor or destructor *function-try-block* handler. Doing so results in undefined behavior.

9.4.1 Noncompliant Code Example

In this noncompliant code example, the constructor for class `C` handles exceptions with a *function-try-block*. However, it generates undefined behavior by inspecting its member field `str`.

```
#include <string>

class C {
  std::string str;

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

9.4.2 Compliant Solution

In this compliant solution, the handler inspects the constructor parameter rather than the class data member, thereby avoiding undefined behavior.

```
#include <string>

class C {
  std::string str;

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

9.4.3 Risk Assessment

Accessing nonstatic data in a constructor's exception handler or a destructor's exception handler leads to undefined behavior.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
ERR53-CPP	Low	Unlikely	Medium	**P2**	**L3**

9.4.4 Related Guidelines

[MISRA 2008]	Rule 15-3-3 (Required)

9.4.5 Bibliography

[ISO/IEC 14882-2014]	Subclause 15.3, "Handling an Exception"

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

The C++ Standard, [except.handle], paragraph 4 [ISO/IEC 14882-2014], states the following:

> The handlers for a try block are tried in order of appearance. That makes it possible to write handlers that can never be executed, for example by placing a handler for a derived class after a handler for a corresponding base class.

Consequently, if two handlers catch exceptions that are derived from the same base class (such as `std::exception`), the most derived exception must come first.

9.5.1 Noncompliant Code Example

In this noncompliant code example, the first handler catches all exceptions of class `B`, as well as exceptions of class `D`, since they are also of class `B`. Consequently, the second handler does not catch any exceptions.

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

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

9.5.2 Compliant Solution

In this compliant solution, the first handler catches all exceptions of class `D`, and the second handler catches all the other exceptions of class `B`.

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

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

9.5.3 Risk Assessment

Exception handlers with inverted priorities cause unexpected control flow when an exception of the derived type occurs.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
ERR54-CPP	Medium	Likely	Low	**P18**	**L1**

9.5.4 Related Guidelines

[MISRA 2008]	Rule 15-3-6 (Required) Rule 15-3-7 (Required)

9.5.5 Bibliography

[ISO/IEC 14882-2014]	Subclause 15.3, “Handling an Exception”

9.6 ERR55-CPP. Honor exception specifications

The C++ Standard, [except.spec], paragraph 8 [ISO/IEC 14882-2014], states the following:

> A function is said to *allow* an exception of type E if the *constant-expression* in its *noexcept-specification* evaluates to `false` or its *dynamic-exception-specification* contains a type T for which a handler of type T would be a match (15.3) for an exception of type E.

If a function throws an exception other than one allowed by its *exception-specification*, it can lead to an implementation-defined termination of the program ([except.spec], paragraph 9).

If a function declared with a *dynamic-exception-specification* throws an exception of a type that would not match the *exception-specification*, the function `std::unexpected()` is called. The behavior of this function can be overridden but, by default, causes an exception of `std::bad_exception` to be thrown. Unless `std::bad_exception` is listed in the *exception-specification*, the function `std::terminate()` will be called.

Similarly, if a function declared with a *noexcept-specification* throws an exception of a type that would cause the *noexcept-specification* to evaluate to `false`, the function `std::terminate()` will be called.

Calling `std::terminate()` leads to implementation-defined termination of the program. To prevent abnormal termination of the program, any function that declares an *exception-specification* should restrict itself, as well as any functions it calls, to throwing only allowed exceptions.

9.6.1 Noncompliant Code Example

In this noncompliant code example, the second function claims to throw only `Exception1`, but it may also throw `Exception2`.

```
#include <exception>

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

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

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

9.6.2 Compliant Solution

This compliant solution catches the exceptions thrown by `foo()`.

```
#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
  }
}
```

9.6.3 Compliant Solution

This compliant solution declares a dynamic *exception-specification* for `bar()`, which covers all of the exceptions that can be thrown from it.

```
#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();
}
```

9.6.4 Noncompliant Code Example

In this noncompliant code example, a function is declared as nonthrowing, but it is possible for `std::vector::resize()` to throw an exception when the requested memory cannot be allocated.

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

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

9.6.5 Compliant Solution

In this compliant solution, the function's *noexcept-specification* is removed, signifying that the function allows all exceptions.

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

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

9.6.5.1 Implementation Details

Some vendors provide language extensions for specifying whether or not a function throws. For instance, Microsoft Visual Studio provides `__declspec(nothrow))`, and Clang supports `__attribute__((nothrow))`. Currently, the vendors do not document the behavior of specifying a nonthrowing function using these extensions. Throwing from a function declared with one of these language extensions is presumed to be undefined behavior.

9.6.6 Risk Assessment

Throwing unexpected exceptions disrupts control flow and can cause premature termination and denial of service.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
ERR55-CPP	Low	Likely	Low	**P9**	**L2**

9.6.7 Related Guidelines

SEI CERT C++ Coding Standard	ERR50-CPP. Do not abruptly terminate the program

9.6.8 Bibliography

[GNU 2016]	"Declaring Attributes of Functions"
[ISO/IEC 14882-2014]	Subclause 15.4, "Exception Specifications"
[MSDN 2016]	"`nothrow` (C++)"

9.7 ERR56-CPP. Guarantee exception safety

Proper handling of errors and exceptional situations is essential for the continued correct operation of software. The preferred mechanism for reporting errors in a C++ program is exceptions rather than error codes. A number of core language facilities, including `dynamic_cast`, `operator new()`, and `typeid`, report failures by throwing exceptions. In addition, the C++ standard library makes heavy use of exceptions to report several different kinds of failures. Few C++ programs manage to avoid using some of these facilities. Consequently, the vast majority of C++ programs must be prepared for exceptions to occur and must handle each appropriately. (See ERR51-CPP. Handle all exceptions.)

Because exceptions introduce code paths into a program, it is important to consider the effects of code taking such paths and to avoid any undesirable effects that might arise otherwise. Some such effects include failure to release an acquired resource, thereby introducing a leak, and failure to reestablish a class invariant after a partial update to an object or even a partial object update while maintaining all invariants. Code that avoids any such undesirable effects is said to be exception safe.

Based on the preceding effects, the following table distinguishes three kinds of exception safety guarantees from most to least desired.

Guarantee	Description	Example
Strong	The strong exception safety guarantee is a property of an operation such that, in addition to satisfying the basic exception safety guarantee, if the operation terminates by raising an exception, it has no observable effects on program state.	Strong Exception Safety
Basic	The basic exception safety guarantee is a property of an operation such that, if the operation terminates by raising an exception, it preserves program state invariants and prevents resource leaks.	Basic Exception Safety
None	Code that provides neither the strong nor basic exception safety guarantee is not exception safe.	No Exception Safety

Code that guarantees strong exception safety also guarantees basic exception safety.

Because all exceptions thrown in an application must be handled, in compliance with ERR50-CPP. Do not abruptly terminate the program, it is critical that thrown exceptions do not leave the program in an indeterminate state where invariants are violated. That is, the program must provide basic exception safety for all invariants and may choose to provide strong exception safety for some invariants. Whether exception handling is used to control the termination of the program or to recover from an exceptional situation, a violated invariant leaves the program in a state where graceful continued execution is likely to introduce security vulnerabilities. Thus, code that provides no exception safety guarantee is unsafe and must be considered defective.

9.7.1 Noncompliant Code Example (No Exception Safety)

The following noncompliant code example shows a flawed copy assignment operator. The implicit invariants of the class are that the `array` member is a valid (possibly null) pointer and that the `nElems` member stores the number of elements in the array pointed to by `array`. The function deallocates `array` and assigns the element counter, `nElems`, before allocating a new block of memory for the copy. As a result, if the `new` expression throws an exception, the function will have modified the state of both member variables in a way that violates the implicit invariants of the class. Consequently, such an object is in an indeterminate state and any operation on it, including its destruction, results in undefined behavior.

```
#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;
  }

  // ...
};
```

9.7.2 Compliant Solution (Strong Exception Safety)

In this compliant solution, the copy assignment operator provides the strong exception safety guarantee. The function allocates new storage for the copy before changing the state of the object. Only after the allocation succeeds does the function proceed to change the state of the object. In addition, by copying the array to the newly allocated storage before deallocating the existing array, the function avoids the test for self-assignment, which improves the performance of the code in the common case [Sutter 2004].

```
#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;
  }

  // ...
};
```

9.7.3 Risk Assessment

Code that is not exception safe typically leads to resource leaks, causes the program to be left in an inconsistent or unexpected state, and ultimately results in undefined behavior at some point after the first exception is thrown.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
ERR56-CPP	High	Likely	High	**P9**	**L2**

9.7.4 Related Guidelines

SEI CERT C++ Coding Standard	ERR51-CPP. Handle all exceptions
MITRE CWE	CWE-703, Failure to Handle Exceptional Conditions CWE-754, Improper Check for Unusual or Exceptional Conditions CWE-755, Improper Handling of Exceptional Conditions

9.7.5 Bibliography

[ISO/IEC 14882-2014]	Clause 15, "Exception Handling"
[Stroustrup 2001]	
[Sutter 2000]	
[Sutter 2001]	
[Sutter 2004]	55. "Prefer the canonical form of assignment."

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

Reclaiming resources when exceptions are thrown is important. An exception being thrown may result in cleanup code being bypassed or an object being left in a partially initialized state. Such a partially initialized object would violate basic exception safety, as described in ERR56-CPP. Guarantee exception safety. It is preferable that resources be reclaimed automatically, using the RAII design pattern [Stroustrup 2001], when objects go out of scope. This technique avoids the need to write complex cleanup code when allocating resources.

However, constructors do not offer the same protection. Because a constructor is involved in allocating resources, it does not automatically free any resources it allocates if it terminates prematurely. The C++ Standard, [except.ctor], paragraph 2 [ISO/IEC 14882-2014], states the following:

> An object of any storage duration whose initialization or destruction is terminated by an exception will have destructors executed for all of its fully constructed subobjects (excluding the variant members of a union-like class), that is, for subobjects for which the principal constructor (12.6.2) has completed execution and the destructor has not yet begun execution. Similarly, if the non-delegating constructor for an object has completed execution and a delegating constructor for that object exits with an exception, the object's destructor will be invoked. If the object was allocated in a new-expression, the matching deallocation function (3.7.4.2, 5.3.4, 12.5), if any, is called to free the storage occupied by the object.

It is generally recommended that constructors that cannot complete their job should throw exceptions rather than exit normally and leave their object in an incomplete state [Cline 2009].

Resources must not be leaked as a result of throwing an exception, including during the construction of an object.

This rule is a subset of MEM51-CPP. Properly deallocate dynamically allocated resources, as all failures to deallocate resources violate that rule.

9.8.1 Noncompliant Code Example

In this noncompliant code example, `pst` is not properly released when `process_item` throws an exception, causing a resource leak.

```
#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;
}
```

9.8.2 Compliant Solution (`delete`)

In this compliant solution, the exception handler frees `pst` by calling `delete`.

```
#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;
}
```

While this compliant solution properly releases its resources using `catch` clauses, this approach can have some disadvantages:

- Each distinct cleanup requires its own `try` and `catch` blocks.
- The cleanup operation must not throw any exceptions.

9.8.3 Compliant Solution (RAII Design Pattern)

A better approach is to employ RAII. This pattern forces every object to clean up after itself in the face of abnormal behavior, preventing the programmer from having to do so. Another benefit of this approach is that it does not require statements to handle resource allocation errors, in conformance with MEM52-CPP. Detect and handle memory allocation errors.

```
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.
```

9.8.4 Noncompliant Code Example

In this noncompliant code example, the `C::C()` constructor might fail to allocate memory for `a`, might fail to allocate memory for `b`, or might throw an exception in the `init()` method. If `init()` throws an exception, neither `a` nor `b` will be released. Likewise, if the allocation for `b` fails, `a` will not be released.

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

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

9.8.5 Compliant Solution (`try/catch`)

This compliant solution mitigates the potential failures by releasing `a` and `b` if an exception is thrown during their allocation or during `init()`.

```
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;
    }
  }
};
```

9.8.6 Compliant Solution (std::unique_ptr)

This compliant solution uses `std::unique_ptr` to create objects that clean up after themselves should anything go wrong in the `C::C()` constructor. The `std::unique_ptr` applies the principles of RAII to pointers.

```
#include <memory>

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

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

9.8.7 Risk Assessment

Memory and other resource leaks will eventually cause a program to crash. If an attacker can provoke repeated resource leaks by forcing an exception to be thrown through the submission of suitably crafted data, then the attacker can mount a denial-of-service attack.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
ERR57-CPP	Low	Probable	High	**P2**	**L3**

9.8.8 Related Guidelines

SEI CERT C++ Coding Standard	MEM51-CPP. Properly deallocate dynamically allocated resources MEM52-CPP. Detect and handle memory allocation errors ERR56-CPP. Guarantee exception safety

9.8.9 Bibliography

[Cline 2009]	Question 17.2, I'm still not convinced: A 4-line code snippet shows that return-codes aren't any worse than exceptions; why should I therefore use exceptions on an application that is orders of magnitude larger?
[ISO/IEC 14882-2014]	Subclause 15.2, "Constructors and Destructors"
[Meyers 1996]	Item 9, "Use Destructors to Prevent Resource Leaks"
[Stroustrup 2001]	"Exception-Safe Implementation Techniques"

9.9 ERR58-CPP. Handle all exceptions thrown before main() begins executing

Not all exceptions can be caught, even with careful use of *function-try-blocks*. The C++ Standard, [except.handle], paragraph 13 [ISO/IEC 14882-2014], states the following:

> Exceptions thrown in destructors of objects with static storage duration or in constructors of namespace scope objects with static storage duration are not caught by a *function-try-block* on `main()`. Exceptions thrown in destructors of objects with thread storage duration or in constructors of namespace-scope objects with thread storage duration are not caught by a function-try-block on the initial function of the thread.

When declaring an object with static or thread storage duration, and that object is not declared within a function block scope, the type's constructor must be declared `noexcept` and must comply with ERR55-CPP. Honor exception specifications. Additionally, the initializer for such a declaration, if any, must not throw an uncaught exception (including from any implicitly constructed objects that are created as a part of the initialization). If an uncaught exception is thrown before `main()` is executed, or if an uncaught exception is thrown after `main()` has finished executing, there are no further opportunities to handle the exception and it results in implementation-defined behavior. (See ERR50-CPP. Do not abruptly terminate the program for further details.)

For more information on exception specifications of destructors, see DCL57-CPP. Do not let exceptions escape from destructors or deallocation functions.

9.9.1 Noncompliant Code Example

In this noncompliant example, the constructor for S may throw an exception that is not caught when `globalS` is constructed during program startup.

```
struct S {
  S() noexcept(false);
};
static S globalS;
```

9.9.2 Compliant Solution

This compliant solution makes `globalS` into a local variable with static storage duration, allowing any exceptions thrown during object construction to be caught because the constructor for `S` will be executed the first time the function `globalS()` is called rather than at program startup. This solution does require the programmer to modify source code so that previous uses of `globalS` are replaced by a function call to `globalS()`.

```
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.
}
```

9.9.3 Noncompliant Code Example

In this noncompliant example, the constructor of `global` may throw an exception during program startup. (The `std::string` constructor, which accepts a `const char *` and a default allocator object, is not marked `noexcept` and consequently allows all exceptions.) This exception is not caught by the *function-try-block* on `main()`, resulting in a call to `std::terminate()` and abnormal program termination.

```
#include <string>

static const std::string global("...");

int main()
try {
  // ...
} catch(...) {
  // IMPORTANT: Will not catch exceptions thrown
  // from the constructor of global
}
```

9.9.4 Compliant Solution

Compliant code must prevent exceptions from escaping during program startup and termination. This compliant solution avoids defining a `std::string` at global namespace scope and instead uses a `static const char *`.

```
static const char *global = "...";

int main() {
  // ...
}
```

9.9.5 Compliant Solution

This compliant solution introduces a class derived from `std::string` with a constructor that catches all exceptions with a function try block and terminates the application in accordance with **ERR50-CPP-EX1** in ERR50-CPP. Do not abruptly terminate the program in the event any exceptions are thrown. Because no exceptions can escape the constructor, it is marked `noexcept` and the class type is permissible to use in the declaration or initialization of a static global variable.

For brevity, the full interface for such a type is not described.

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

namespace my {
struct string : std::string {
  explicit string(const char *msg,
                  const std::string::allocator_type &alloc =
std::string::allocator_type{}) noexcept
  try : std::string(msg, alloc) {} catch(...) {
    extern void log_message(const char *) noexcept;
    log_message("std::string constructor threw an exception");
    std::terminate();
  }
  // ...
};
}

static const my::string global("...");

int main() {
  // ...
}
```

9.9.6 Noncompliant Code Example

In this noncompliant example, an exception may be thrown by the initializer for the static global variable `i`.

```
extern int f() noexcept(false);
int i = f();

int main() {
  // ...
}
```

9.9.7 Compliant Solution

This compliant solution wraps the call to `f()` with a helper function that catches all exceptions and terminates the program in conformance with **ERR50-CPP-EX1** of ERR50-CPP. Do not abruptly terminate the program.

```
#include <exception>

int f_helper() noexcept {
  try {
    extern int f() noexcept(false);
    return f();
  } catch (...) {
    extern void log_message(const char *) noexcept;
    log_message("f() threw an exception");
    std::terminate();
  }
  // Unreachable.
}

int i = f_helper();

int main() {
  // ...
}
```

9.9.8 Risk Assessment

Throwing an exception that cannot be caught results in abnormal program termination and can lead to denial-of-service attacks.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
ERR58-CPP	Low	Likely	Low	**P9**	**L2**

9.9.9 Related Guidelines

This rule is a subset of ERR50-CPP. Do not abruptly terminate the program.

SEI CERT C++ Coding Standard	DCL57-CPP. Do not let exceptions escape from destructors or deallocation functions ERR55-CPP. Honor exception specifications

9.9.10 Bibliography

[ISO/IEC 14882-2014]	Subclause 15.4, “Exception Specifications”
[Sutter 2000]	Item 8, “Writing Exception-Safe Code—Part 1”

9.10 ERR59-CPP. Do not throw an exception across execution boundaries

Throwing an exception requires collaboration between the execution of the `throw` expression and the passing of control to the appropriate `catch` statement, if one applies. This collaboration takes the form of runtime logic used to calculate the correct handler for the exception and is an implementation detail specific to the platform. For code compiled by a single C++ compiler, the details of how to throw and catch exceptions can be safely ignored. However, when throwing an exception across execution boundaries, care must be taken to ensure the runtime logic used is compatible between differing sides of the execution boundary.

An *execution boundary* is the delimitation between code compiled by differing compilers, including different versions of a compiler produced by the same vendor. For instance, a function may be declared in a header file but defined in a library that is loaded at runtime. The execution boundary is between the call site in the executable and the function implementation in the library. Such boundaries are also called ABI (application binary interface) boundaries because they relate to the interoperability of application binaries.

Throw an exception across an execution boundary only when both sides of the execution boundary use the same ABI for exception handling.

9.10.1 Noncompliant Code Example

In this noncompliant code example, an exception is thrown from a library function to signal an error. Despite the exception being a scalar type (and thus implicitly conforming to EXP60-CPP. Do not pass a nonstandard-layout type object across execution boundaries), this code can still result in abnormal program execution if the library and application adhere to different ABIs.

```
// 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
  }
}
```

9.10.1.1 Implementation Details

If the library code is compiled (with modification to account for mangling differences) with GCC 4.9 on a default installation of MinGW-w64 without special compiler flags, the exception throw will rely on the zero-cost, table-based exception model that is based on DWARF and uses the Itanium ABI. If the application code is compiled with Microsoft Visual Studio 2013, the catch handler will be based on Structured Exception Handling and the Microsoft ABI. These two exception-handling formats are incompatible, as are the ABIs, resulting in abnormal program behavior. Specifically, the exception thrown by the library is not caught by the application, and `std::terminate()` is eventually called.

9.10.2 Compliant Solution

In this compliant solution, the error from the library function is indicated by a return value instead of an exception. Using Microsoft Visual Studio (or GCC) to compile both the library and the application would also be a compliant solution because the same exception-handling machinery and ABI would be used on both sides of the execution boundary.

```
// library.h
int func() noexcept(true); // Implemented by the library

// library.cpp
int func() noexcept(true) {
  // ...
  if (/* ... */) {
    return 42;
  }
  // ...
  return 0;
}

// application.cpp
#include "library.h"

void f() {
  if (int err = func()) {
    // Handle error
  }
}
```

9.10.3 Risk Assessment

The effects of throwing an exception across execution boundaries depends on the implementation details of the exception-handling mechanics. They can range from correct or benign behavior to undefined behavior.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
ERR59-CPP	High	Probable	Medium	**P12**	**L1**

9.10.4 Related Guidelines

CERT C++ Coding Standard	EXP60-CPP. Do not pass a nonstandard-layout type object across execution boundaries

9.10.5 Bibliography

[ISO/IEC 14882-2014]	Subclause 15, “Exception Handling”

9.11 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 undefined 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.

9.11.1 Noncompliant Code Example

In this noncompliant code example, an exception of type `S` is thrown from `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.

```
#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
  }
}
```

9.11.2 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
  }
}
```

9.11.3 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
  }
}
```

9.11.4 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
ERR60-CPP	Low	Probable	Medium	**P4**	**L3**

9.11.5 Related Guidelines

SEI CERT C++ Coding Standard	ERR50-CPP. Do not abruptly terminate the program ERR55-CPP. Honor exception specifications

9.11.6 Bibliography

[Hinnant 2015]	
[ISO/IEC 14882-2014]	Subclause 15.1, “Throwing an Exception” Subclause 18.8.1, “Class `exception`” Subclause 18.8.5, “Exception Propagation”

9.12 ERR61-CPP. Catch exceptions by lvalue reference

When an exception is thrown, the value of the object in the throw expression is used to initialize an anonymous temporary object called the *exception object*. The type of this exception object is used to transfer control to the nearest catch handler, which contains an exception declaration with a matching type. The C++ Standard, [except.handle], paragraph 16 [ISO/IEC 14882-2014], in part, states the following:

> The variable declared by the *exception-declaration*, of type *cv* T or *cv* T&, is initialized from the exception object, of type E, as follows:
>
> - if T is a base class of E, the variable is copy-initialized from the corresponding base class subobject of the exception object;
> - otherwise, the variable is copy-initialized from the exception object.

Because the variable declared by the *exception-declaration* is copy-initialized, it is possible to *slice* the exception object as part of the copy operation, losing valuable exception information and leading to incorrect error recovery. For more information about object slicing, see OOP51-CPP. Do not slice derived objects. Further, if the copy constructor of the exception object throws an exception, the copy initialization of the *exception-declaration* object results in undefined behavior. (See ERR60-CPP. Exception objects must be nothrow copy constructible for more information.)

Always catch exceptions by lvalue reference unless the type is a trivial type. For reference, the C++ Standard, [basic.types], paragraph 9 [ISO/IEC 14882-2014], defines trivial types as the following:

> Arithmetic types, enumeration types, pointer types, pointer to member types, `std::nullptr_t`, and cv-qualified versions of these types are collectively called *scalar types*.... Scalar types, trivial class types, arrays of such types and cv-qualified versions of these types are collectively called *trivial types*.

The C++ Standard, [class], paragraph 6, defines trivial class types as the following:

> A *trivially copyable class* is a class that:
>
> - has no non-trivial copy constructors,
> - has no non-trivial move constructors,
> - has no non-trivial copy assignment operators,
> - has no non-trivial move assignment operators, and
> - has a trivial destructor.
>
> A *trivial class* is a class that has a default constructor, has no non-trivial default constructors, and is trivially copyable. [*Note*: In particular, a trivially copyable or trivial class does not have virtual functions or virtual base classes. — *end note*]

9.12.1 Noncompliant Code Example

In this noncompliant code example, an object of type `S` is used to initialize the exception object that is later caught by an *exception-declaration* of type `std::exception`. The *exception-declaration* matches the exception object type, so the variable `E` is copy-initialized from the exception object, resulting in the exception object being sliced. Consequently, the output of this noncompliant code example is the implementation-defined value returned from calling `std::exception::what()` instead of `"My custom exception"`.

```
#include <exception>
#include <iostream>

struct S : std::exception {
  const char *what() const noexcept override {
    return "My custom exception";
  }
};

void f() {
  try {
    throw S();
  } catch (std::exception e) {
    std::cout << e.what() << std::endl;
  }
}
```

9.12.2 Compliant Solution

In this compliant solution, the variable declared by the *exception-declaration* is an lvalue reference. The call to `what()` results in executing `S::what()` instead of `std::exception::what()`.

```
#include <exception>
#include <iostream>

struct S : std::exception {
  const char *what() const noexcept override {
    return "My custom exception";
  }
};

void f() {
  try {
    throw S();
  } catch (std::exception &e) {
    std::cout << e.what() << std::endl;
  }
}
```

9.12.3 Risk Assessment

Object slicing can result in abnormal program execution. This generally is not a problem for exceptions, but it can lead to unexpected behavior depending on the assumptions made by the exception handler.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
ERR61-CPP	Low	Unlikely	Low	**P3**	**L3**

9.12.4 Related Guidelines

This rule is a subset of OOP51-CPP. Do not slice derived objects.

SEI CERT C++ Coding Standard	ERR60-CPP. Exception objects must be nothrow copy constructible

9.12.5 Bibliography

[ISO/IEC 14882-2014]	Subclause 3.9, “Types” Clause 9, “Classes” Subclause 15.1, “Throwing an Exception” Subclause 15.3, “Handling an Exception”
[MISRA 2008]	Rule 15-3-5

9.13 ERR62-CPP. Detect errors when converting a string to a number

The process of parsing an integer or floating-point number from a string can produce many errors. The string might not contain a number. It might contain a number of the correct type that is out of range (such as an integer that is larger than `INT_MAX`). The string may also contain extra information after the number, which may or may not be useful after the conversion. These error conditions must be detected and addressed when a string-to-number conversion is performed using a formatted input stream such as `std::istream` or the locale facet `num_get<>`.

When calling a formatted input stream function like `istream::operator>>()`, information about conversion errors is queried through the `basic_ios::good()`, `basic_ios::bad()`, and `basic_ios::fail()` inherited member functions or through exception handling if it is enabled on the stream object.

When calling `num_get<>::get()`, information about conversion errors is returned to the caller through the `ios_base::iostate&` argument. The C++ Standard, section [facet.num.get.virtuals], paragraph 3 [ISO/IEC 14882-2014], in part, states the following:

> If the conversion function fails to convert the entire field, or if the field represents a value outside the range of representable values, `ios_base::failbit` is assigned to `err`.

Always explicitly check the error state of a conversion from string to a numeric value (or handle the related exception, if applicable) instead of assuming the conversion results in a valid value. This rule is in addition to ERR34-C. Detect errors when converting a string to a number, which bans the use of conversion functions that do not perform conversion validation such as `std::atoi()` and `std::scanf()` from the C Standard Library.

9.13.1 Noncompliant Code Example

In this noncompliant code example, multiple numeric values are converted from the standard input stream. However, if the text received from the standard input stream cannot be converted into a numeric value that can be represented by an `int`, the resulting value stored into the variables `i` and `j` may be unexpected.

```
#include <iostream>

void f() {
  int i, j;
  std::cin >> i >> j;
  // ...
}
```

For instance, if the text `12345678901234567890` is the first converted value read from the standard input stream, then `i` will have the value `std::numeric_limits<int>::max()` (per [facet.num.get.virtuals] paragraph 3), and `j` will be uninitialized (per [istream.formatted.arithmetic] paragraph 3). If the text `abcdefg` is the first converted value read from the standard input stream, then `i` will have the value `0` and `j` will remain uninitialized.

9.13.2 Compliant Solution

In this compliant solution, exceptions are enabled so that any conversion failure results in an exception being thrown. However, this approach cannot distinguish between which values are valid and which values are invalid and must assume that all values are invalid. Both the `badbit` and `failbit` flags are set to ensure that conversion errors as well as loss of integrity with the stream are treated as exceptions.

```
#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
  }
}
```

9.13.3 Compliant Solution

In this compliant solution, each converted value read from the standard input stream is tested for validity before reading the next value in the sequence, allowing error recovery on a per-value basis. It checks `std::istream::fail()` to see if the failure bit was set due to a conversion failure or whether the bad bit was set due to a loss of integrity with the stream object. If a failure condition is encountered, it is cleared on the input stream and then characters are read and discarded until a `' '` (space) character occurs. The error handling in this case only works if a space character is what delimits the two numeric values to be converted.

```
#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(), ' ');
  }

  // ...
}
```

9.13.4 Risk Assessment

It is rare for a violation of this rule to result in a security vulnerability unless it occurs in security-sensitive code. However, violations of this rule can easily result in lost or misinterpreted data.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
ERR62-CPP	Medium	Unlikely	Medium	**P4**	**L3**

9.13.5 Related Guidelines

SEI CERT C Coding Standard	ERR34-C. Detect errors when converting a string to a number
MITRE CWE	CWE-676, Use of potentially dangerous function CWE-20, Insufficient input validation

9.13.6 Bibliography

[ISO/IEC 9899:1999]	Subclause 7.22.1, “Numeric conversion functions” Subclause 7.21.6, “Formatted input/output functions”
[ISO/IEC 14882-2014]	Subclause 22.4.2.1.1, “num_get members” Subclause 27.7.2.2, “Formatted input functions”

10 Object Oriented Programming (OOP)

10.1 OOP50-CPP. Do not invoke virtual functions from constructors or destructors

Virtual functions allow for static dispatch of member function calls at runtime based on the dynamic type of the object that the member function is being called on. This convention supports object-oriented programming practices commonly associated with object inheritance and function overriding. When calling a nonvirtual member function or when using a class member access expression to denote a call, the specified function is called. Otherwise, a virtual function call is made to the final overrider in the dynamic type of the object expression.

However, during the construction and destruction of an object, the rules for virtual method dispatch on that object are restricted. The C++ Standard, [class.cdtor], paragraph 4 [ISO/IEC 14882-2014], states the following:

> Member functions, including virtual functions, can be called during construction or destruction. When a virtual function is called directly or indirectly from a constructor or from a destructor, including during the construction or destruction of the class's non-static data members, and the object to which the call applies is the object (call it `x`) under construction or destruction, the function called is the final overrider in the constructor's or destructor's class and not one overriding it in a more-derived class. If the virtual function call uses an explicit class member access and the object expression refers to the complete object of `x` or one of that object's base class subobjects but not `x` or one of its base class subobjects, the behavior is undefined.

Do not directly or indirectly invoke a virtual function from a constructor or destructor that attempts to call into the object under construction or destruction. Because the order of construction starts with base classes and moves to more derived classes, attempting to call a derived class function from a base class under construction is dangerous. The derived class has not had the opportunity to initialize its resources, which is why calling a virtual function from a constructor does not result in a call to a function in a more derived class. Similarly, an object is destroyed in reverse order from construction, so attempting to call a function in a more derived class from a destructor may access resources that have already been released.

10.1.1 Noncompliant Code Example

In this noncompliant code example, the base class attempts to seize and release an object's resources through calls to virtual functions from the constructor and destructor. However, the `B::B()` constructor calls `B::seize()` rather than `D::seize()`. Likewise, the `B::~B()` destructor calls `B::release()` rather than `D::release()`.

```
struct B {
  B() { seize(); }
  virtual ~B() { release(); }

protected:
  virtual void seize();
  virtual void release();
};

struct D : B {
  virtual ~D() = default;

protected:
  void seize() override {
    B::seize();
    // Get derived resources...
  }

  void release() override {
    // Release derived resources...
    B::release();
  }
};
```

The result of running this code is that no derived class resources will be seized or released during the initialization and destruction of object of type `D`. At the time of the call to `seize()` from `B::B()`, the `D` constructor has not been entered, and the behavior of the under-construction object will be to invoke `B::seize()` rather than `D::seize()`. A similar situation occurs for the call to `release()` in the base class destructor. If the functions `seize()` and `release()` were declared to be pure virtual functions, the result would be undefined behavior.

10.1.2 Compliant Solution

In this compliant solution, the constructors and destructors call a nonvirtual, private member function (suffixed with `mine`) instead of calling a virtual function. The result is that each class is responsible for seizing and releasing its own resources.

```
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();
  }
};
```

10.1.3 Exceptions

OOP50-CPP-EX1: Because valid use cases exist that involve calling (non-pure) virtual functions from the constructor of a class, it is permissible to call the virtual function with an explicitly qualified ID. The qualified ID signifies to code maintainers that the expected behavior is for the class under construction or destruction to be the final overrider for the function call.

```
struct A {
  A() {
    // f();   // WRONG!
    A::f();   // Okay
  }
  virtual void f();
};
```

OOP50-CPP-EX2: It is permissible to call a virtual function that has the `final` *virt-specifier* from a constructor or destructor, as in this example.

```
struct A {
  A();
  virtual void f();
};

struct B : A {
  B() : A() {
    f();  // Okay
  }
  void f() override final;
};
```

Similarly, it is permissible to call a virtual function from a constructor or destructor of a class that has the `final` *class-virt-specifier*, as in this example.

```
struct A {
  A();
  virtual void f();
};

struct B final : A {
  B() : A() {
    f();  // Okay
  }
  void f() override;
};
```

In either case, `f()` must be the final overrider, guaranteeing consistent behavior of the function being called.

10.1.4 Risk Assessment

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
OOP50-CPP	Low	Unlikely	Medium	**P2**	**L3**

10.1.5 Bibliography

[Dewhurst 2002]	Gotcha #75, "Calling Virtual Functions in Constructors and Destructors"
[ISO/IEC 14882-2014]	Subclause 5.5, "Pointer-to-Member Operators"
[Lockheed Martin 2005]	AV Rule 71.1, "A class's virtual functions shall not be invoked from its destructor or any of its constructors"
[Sutter 2004]	Item 49, "Avoid Calling Virtual Functions in Constructors and Destructors"

10.2 OOP51-CPP. Do not slice derived objects

An object deriving from a base class typically contains additional member variables that extend the base class. When by-value assigning or copying an object of the derived type to an object of the base type, those additional member variables are not copied because the base class contains insufficient space in which to store them. This action is commonly called *slicing* the object because the additional members are “sliced off” the resulting object.

Do not initialize an object of base class type with an object of derived class type, except through references, pointers, or pointer-like abstractions (such as `std::unique_ptr, or std::shared_ptr`).

10.2.1 Noncompliant Code Example

In this noncompliant code example, an object of the derived `Manager` type is passed by value to a function accepting a base `Employee` type. Consequently, the `Manager` objects are sliced, resulting in information loss and unexpected behavior when the `print()` function is called.

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

class Employee {
  std::string name;

protected:
  virtual void print(std::ostream &os) const {
    os << "Employee: " << get_name() << std::endl;
  }

public:
  Employee(const std::string &name) : name(name) {}
  const std::string &get_name() const { return name; }
  friend std::ostream &operator<<
      (std::ostream &os, const Employee &e) {
    e.print(os);
    return os;
  }
};

class Manager : public Employee {
  Employee assistant;

protected:
  void print(std::ostream &os) const override {
    os << "Manager: " << get_name() << std::endl;
    os << "Assistant: " << std::endl << "\t"
       << get_assistant() << std::endl;
  }
public:
  Manager(const std::string &name, const Employee &assistant) :
      Employee(name), assistant(assistant) {}
  const Employee &get_assistant() const { return assistant; }
};

void f(Employee e) {
  std::cout << e;
}

int main() {
  Employee coder("Joe Smith");
  Employee typist("Bill Jones");
  Manager designer("Jane Doe", typist);

  f(coder);
  f(typist);
  f(designer);
}
```

When `f()` is called with the `designer` argument, the formal parameter in `f()` is sliced and information is lost. When the object `e` is printed, `Employee::print()` is called instead of `Manager::print()`, resulting in the following output:

```
Employee: Jane Doe
```

10.2.2 Compliant Solution (Pointers)

Using the same class definitions as the noncompliant code example, this compliant solution modifies the definition of `f()` to require raw pointers to the object, removing the slicing problem.

```
// 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);
}
```

This compliant solution also complies with EXP34-C. Do not dereference null pointers in the implementation of `f()`. With this definition, the program correctly outputs the following.

```
Employee: Joe Smith
Employee: Bill Jones
Manager: Jane Doe
Assistant:
        Employee: Bill Jones
```

10.2.3 Compliant Solution (References)

An improved compliant solution, which does not require guarding against null pointers within `f()`, uses references instead of pointers.

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

10.2.4 Compliant Solution (Noncopyable)

Both previous compliant solutions depend on consumers of the `Employee` and `Manager` types to be declared in a compliant manner with the expected usage of the class hierarchy. This compliant solution ensures that consumers are unable to accidentally slice objects by removing the ability to copy-initialize an object that derives from `Noncopyable`. If copy-initialization is attempted, as in the original definition of `f()`, the program is ill-formed and a diagnostic will be emitted. However, such a solution also restricts the `Manager` object from attempting to copy-initialize its `Employee` object, which subtly changes the semantics of the class hierarchy.

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

class Noncopyable {
  Noncopyable(const Noncopyable &) = delete;
  void operator=(const Noncopyable &) = delete;

protected:
  Noncopyable() = default;
};

class Employee : Noncopyable {
  // Remainder of the definition is unchanged.
  std::string name;

protected:
  virtual void print(std::ostream &os) const {
    os << "Employee: " << get_name() << std::endl;
  }

public:
  Employee(const std::string &name) : name(name) {}
  const std::string &get_name() const { return name; }
  friend std::ostream &operator<<(std::ostream &os, const Employee
&e) {
    e.print(os);
    return os;
  }
};

class Manager : public Employee {
  const Employee &assistant;
  // Note: The definition of Employee has been modified.

  // Remainder of the definition is unchanged.
protected:
  void print(std::ostream &os) const override {
    os << "Manager: " << get_name() << std::endl;
    os << "Assistant: " << std::endl << "\t" << get_assistant() <<
std::endl;
  }

public:
  Manager(const std::string &name, const Employee &assistant) :
      Employee(name), 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);
}
```

10.2.5 Noncompliant Code Example

This noncompliant code example uses the same class definitions of `Employee` and `Manager` as in the previous noncompliant code example and attempts to store `Employee` objects in a `std::vector`. However, because `std::vector` requires a homogeneous list of elements, slicing occurs.

```
#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);
}
```

10.2.6 Compliant Solution

This compliant solution uses a vector of `std::unique_ptr` objects, which eliminates the slicing problem.

```
#include <iostream>
#include <memory>
#include <string>
#include <vector>

void f(const std::vector<std::unique_ptr<Employee>> &v) {
  for (const auto &e : v) {
    std::cout << *e;
  }
}

int main() {
  std::vector<std::unique_ptr<Employee>> v;

  v.emplace_back(new Employee("Joe Smith"));
  v.emplace_back(new Employee("Bill Jones"));
  v.emplace_back(new Manager("Jane Doe", *v.front()));

  f(v);
}
```

10.2.7 Risk Assessment

Slicing results in information loss, which could lead to abnormal program execution or denial-of-service attacks.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
OOP51-CPP	Low	Probable	Medium	**P4**	**L3**

10.2.8 Related Guidelines

SEI CERT C++ Coding Standard	ERR61-CPP. Catch exceptions by lvalue reference CTR56-CPP. Do not use pointer arithmetic on polymorphic objects
SEI CERT C Coding Standard	EXP34-C. Do not dereference null pointers

10.2.9 Bibliography

[Dewhurst 2002]	Gotcha #38, “Slicing”
[ISO/IEC 14882-2014]	Subclause 12.8, “Copying and Moving Class Objects”
[Sutter 2000]	Item 40, “Object Lifetimes—Part I”

10.3 OOP52-CPP. Do not delete a polymorphic object without a virtual destructor

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

> In the first alternative (*delete object*), if the static type of the object to be deleted is different from its dynamic type, the static type shall be a base class of the dynamic type of the object to be deleted and the static type shall have a virtual destructor or the behavior is undefined. In the second alternative (*delete array*) if the dynamic type of the object to be deleted differs from its static type, the behavior is undefined.

Do not delete an object of derived class type through a pointer to its base class type that has a non-`virtual` destructor. Instead, the base class should be defined with a `virtual` destructor. Deleting an object through a pointer to a type without a `virtual` destructor results in undefined behavior.

10.3.1 Noncompliant Code Example

In this noncompliant example, `b` is a polymorphic pointer type whose static type is `Base *` and whose dynamic type is `Derived *`. When `b` is deleted, it results in undefined behavior because `Base` does not have a `virtual` destructor. The C++ Standard, [class.dtor], paragraph 4 [ISO/IEC 14882-2014] states the following:

> If a class has no user-declared destructor, a destructor is implicitly declared as defaulted. An implicitly declared destructor is an `inline public` member of its class.

The implicitly declared destructor is not declared as `virtual` even in the presence of other `virtual` functions.

```
struct Base {
  virtual void f();
};

struct Derived : Base {};

void f() {
  Base *b = new Derived();
  // ...
  delete b;
}
```

10.3.2 Noncompliant Code Example

In this noncompliant example, the explicit pointer operations have been replaced with a smart pointer object, demonstrating that smart pointers suffer from the same problem as other pointers. Because the default deleter for `std::unique_ptr` calls `delete` on the internal pointer value, the resulting behavior is identical to the previous noncompliant example.

```
#include <memory>

struct Base {
  virtual void f();
};

struct Derived : Base {};

void f() {
  std::unique_ptr<Base> b = std::make_unique<Derived()>();
}
```

10.3.3 Compliant Solution

In this compliant solution, the destructor for `Base` has an explicitly declared `virtual` destructor, ensuring that the polymorphic delete operation results in well-defined behavior.

```
struct Base {
  virtual ~Base() = default;
  virtual void f();
};

struct Derived : Base {};

void f() {
  Base *b = new Derived();
  // ...
  delete b;
}
```

10.3.4 Risk Assessment

Attempting to destruct a polymorphic object that does not have a `virtual` destructor declared results in undefined behavior. In practice, potential consequences include abnormal program termination and memory leaks.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
OOP52-CPP	Low	Likely	Low	**P9**	**L2**

10.3.5 Related Guidelines

SEI CERT C++ Coding Standard	EXP51-CPP. Do not delete an array through a pointer of the incorrect type

10.3.6 Bibliography

[ISO/IEC 14882-2014]	Subclause 5.3.5, “Delete” Subclause 12.4, “Destructors”
[Stroustrup 2006]	“Why Are Destructors Not Virtual by Default?”

10.4 OOP53-CPP. Write constructor member initializers in the canonical order

The member initializer list for a class constructor allows members to be initialized to specified values and for base class constructors to be called with specific arguments. However, the order in which initialization occurs is fixed and does not depend on the order written in the member initializer list. The C++ Standard, [class.base.init], paragraph 11 [ISO/IEC 14882-2014], states the following:

> In a non-delegating constructor, initialization proceeds in the following order:
>
> - First, and only for the constructor of the most derived class, virtual base classes are initialized in the order they appear on a depth-first left-to-right traversal of the directed acyclic graph of base classes, where “left-to-right” is the order of appearance of the base classes in the derived class base-specifier-list.
> - Then, direct base classes are initialized in declaration order as they appear in the base-specifier-list (regardless of the order of the mem-initializers).
> - Then, non-static data members are initialized in the order they were declared in the class definition (again regardless of the order of the mem-initializers).
> - Finally, the compound-statement of the constructor body is executed.
>
> [Note: The declaration order is mandated to ensure that base and member subobjects are destroyed in the reverse order of initialization. —end note]

Consequently, the order in which member initializers appear in the member initializer list is irrelevant. The order in which members are initialized, including base class initialization, is determined by the declaration order of the class member variables or the base class specifier list. Writing member initializers other than in canonical order can result in undefined behavior, such as reading uninitialized memory.

Always write member initializers in a constructor in the canonical order: first, direct base classes in the order in which they appear in the *base-specifier-list* for the class, then nonstatic data members in the order in which they are declared in the class definition.

10.4.1 Noncompliant Code Example

In this noncompliant code example, the member initializer list for `C::C()` attempts to initialize `someVal` first and then to initialize `dependsOnSomeVal` to a value dependent on `someVal`. Because the declaration order of the member variables does not match the member initializer order, attempting to read the value of `someVal` results in an unspecified value being stored into `dependsOnSomeVal`.

```
class C {
  int dependsOnSomeVal;
  int someVal;

public:
  C(int val) : someVal(val), dependsOnSomeVal(someVal + 1) {}
};
```

10.4.2 Compliant Solution

This compliant solution changes the declaration order of the class member variables so that the dependency can be ordered properly in the constructor's member initializer list.

```
class C {
  int someVal;
  int dependsOnSomeVal;

public:
  C(int val) : someVal(val), dependsOnSomeVal(someVal + 1) {}
};
```

It is reasonable for initializers to depend on previously initialized values.

10.4.3 Noncompliant Code Example

In this noncompliant code example, the derived class, `D`, attempts to initialize the base class, `B1`, with a value obtained from the base class, `B2`. However, because `B1` is initialized before `B2` due to the declaration order in the base class specifier list, the resulting behavior is undefined.

```
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()) {}
};
```

10.4.4 Compliant Solution

This compliant solution initializes both base classes using the same value from the constructor's parameter list instead of relying on the initialization order of the base classes.

```
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) {}
};
```

10.4.5 Risk Assessment

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
OOP53-CPP	Medium	Unlikely	Medium	**P4**	**L3**

10.4.6 Bibliography

[ISO/IEC 14882-2014]	Subclause 12.6.2, "Initializing Bases and Members"
[Lockheed Martin 2005]	AV Rule 75, Members of the initialization list shall be listed in the order in which they are declared in the class

10.5 OOP54-CPP. Gracefully handle self-copy assignment

Self-copy assignment can occur in situations of varying complexity, but essentially, all self-copy assignments entail some variation of the following.

```
#include <utility>

struct S { /* ... */ }

void f() {
  S s;
  s = s; // Self-copy assignment
}
```

User-provided copy operators must properly handle self-copy assignment.

The postconditions required for copy assignment are specified by the C++ Standard, [utility.arg.requirements], Table 23 [ISO/IEC 14882-2014], which states that for `x = y`, the value of `y` is unchanged. When `&x == &y`, this postcondition translates into the values of both `x` and `y` remaining unchanged. A naive implementation of copy assignment could destroy object-local resources in the process of copying resources from the given parameter. If the given parameter is the same object as the local object, the act of destroying object-local resources will invalidate them. The subsequent copy of those resources will be left in an indeterminate state, which violates the postcondition.

A user-provided copy assignment operator must prevent self-copy assignment from leaving the object in an indeterminate state. This can be accomplished by self-assignment tests, copy-and-swap, or other idiomatic design patterns.

The C++ Standard, [copyassignable], specifies that types must ensure that self-copy assignment leave the object in a consistent state when passed to Standard Template Library (STL) functions. Since objects of STL types are used in contexts where `CopyAssignable` is required, STL types are required to gracefully handle self-copy assignment.

10.5.1 Noncompliant Code Example

In this noncompliant code example, the copy assignment operator does not protect against self-copy assignment. If self-copy assignment occurs, `this->s1` is deleted, which results in `rhs.s1` also being deleted. The invalidated memory for `rhs.s1` is then passed into the copy constructor for `S`, which can result in dereferencing an invalid pointer.

```
#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;
  }
};
```

10.5.2 Compliant Solution (Self-Test)

This compliant solution guards against self-copy assignment by testing whether the given parameter is the same as `this`. If self-copy assignment occurs, then `operator=` does nothing; otherwise, the copy proceeds as in the original example.

```
#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;
  }
};
```

This solution does not provide a strong exception guarantee for the copy assignment. Specifically, if an exception is called when evaluating the `new` expression, `this` has already been modified. However, this solution does provide a basic exception guarantee because no resources are leaked and all data members contain valid values. Consequently, this code complies with ERR56-CPP. Guarantee exception safety.

10.5.3 Compliant Solution (Copy and Swap)

This compliant solution avoids self-copy assignment by constructing a temporary object from `rhs` that is then swapped with `*this`. This compliant solution provides a strong exception guarantee because `swap()` will never be called if resource allocation results in an exception being thrown while creating the temporary object.

```
#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;
  }
};
```

10.5.4 Risk Assessment

Allowing a copy assignment operator to corrupt an object could lead to undefined behavior.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
OOP54-CPP	Low	Probable	High	**P2**	**L3**

10.5.5 Related Guidelines

This rule is a partial subset of OOP58-CPP. Copy operations must not mutate the source object when copy operations do not gracefully handle self-copy assignment, because the copy operation may mutate both the source and destination objects (due to them being the same object).

10.5.6 Bibliography

[Henricson 1997]	Rule 5.12, Copy assignment operators should be protected from doing destructive actions if an object is assigned to itself
[ISO/IEC 14882-2014]	Subclause 17.6.3.1, “Template Argument Requirements” Subclause 17.6.4.9, “Function Arguments”
[Meyers 2005]	Item 11, “Handle Assignment to Self in `operator=`”
[Meyers 2014]	

10.6 OOP55-CPP. Do not use pointer-to-member operators to access nonexistent members

The pointer-to-member operators `.*` and `->*` are used to obtain an object or a function as though it were a member of an underlying object. For instance, the following are functionally equivalent ways to call the member function `f()` on the object `o`.

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

void func() {
  S o;
  void (S::*pm)() = &S::f;

  o.f();
  (o.*pm)();
}
```

The call of the form `o.f()` uses class member access at compile time to look up the address of the function `S::f()` on the object `o`. The call of the form `(o.*pm)()` uses the pointer-to-member operator `.*` to call the function at the address specified by `pm`. In both cases, the object `o` is the implicit `this` object within the member function `S::f()`.

The C++ Standard, [expr.mptr.oper], paragraph 4 [ISO/IEC 14882-2014], states the following:

> Abbreviating *pm-expression.*cast-expression* as `E1.*E2`, `E1` is called the *object expression*. If the dynamic type of `E1` does not contain the member to which `E2` refers, the behavior is undefined.

A pointer-to-member expression of the form `E1->*E2` is converted to its equivalent form, `(*(E1)).*E2`, so use of pointer-to-member expressions of either form behave equivalently in terms of undefined behavior.

Further, the C++ Standard, [expr.mptr.oper], paragraph 6, in part, states the following:

> If the second operand is the null pointer to member value, the behavior is undefined.

Do not use a pointer-to-member expression where the dynamic type of the first operand does not contain the member to which the second operand refers, including the use of a null pointer-to-member value as the second operand.

10.6.1 Noncompliant Code Example

In this noncompliant code example, a pointer-to-member object is obtained from `D::g` but is then upcast to be a `B::*`. When called on an object whose dynamic type is `D`, the pointer-to-member call is well defined. However, the dynamic type of the underlying object is `B`, which results in undefined behavior.

```
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;
}
```

10.6.2 Compliant Solution

In this compliant solution, the upcast is removed, rendering the initial code ill-formed and emphasizing the underlying problem that `B::g()` does not exist. This compliant solution assumes that the programmer's intention was to use the correct dynamic type for the underlying object.

```
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;
}
```

10.6.3 Noncompliant Code Example

In this noncompliant code example, a null pointer-to-member value is passed as the second operand to a pointer-to-member expression, resulting in undefined behavior.

```
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;
}
```

10.6.4 Compliant Solution

In this compliant solution, `gptr` is properly initialized to a valid pointer-to-member value instead of to the default value of `nullptr`.

```
struct B {
  virtual ~B() = default;
};

struct D : B {
  virtual ~D() = default;
  virtual void g() { /* ... */ }
};

static void (D::*gptr)() = &D::g; // Explicitly initialized.
void call_memptr(D *ptr) {
  (ptr->*gptr)();
}

void f() {
  D *d = new D;
  call_memptr(d);
  delete d;
}
```

10.6.5 Risk Assessment

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
OOP55-CPP	High	Probable	High	**P6**	**L2**

10.6.6 Related Guidelines

This rule is a subset of EXP34-C. Do not dereference null pointers.

10.6.7 Bibliography

[ISO/IEC 14882-2014]	Subclause 5.5, “Pointer-to-Member Operators”

10.7 OOP56-CPP. Honor replacement handler requirements

The *handler* functions `new_handler`, `terminate_handler`, and `unexpected_handler` can be globally replaced by custom implementations, as specified by [handler.functions], paragraph 2, of the C++ Standard [ISO/IEC 14882-2014]. For instance, an application could set a custom termination handler by calling `std::set_terminate()`, and the custom termination handler may log the termination for later auditing. However, the C++ Standard, [res.on.functions], paragraph 1, states the following:

> In certain cases (replacement functions, handler functions, operations on types used to instantiate standard library template components), the C++ standard library depends on components supplied by a C++ program. If these components do not meet their requirements, the Standard places no requirements on the implementation.

Paragraph 2, in part, further states the following:

> In particular, the effects are undefined in the following cases:
>
> - for handler functions, if the installed handler function does not implement the semantics of the applicable *Required behavior:* paragraph

A replacement for any of the handler functions must meet the semantic requirements specified by the appropriate *Required behavior:* clause of the replaced function.

10.7.1.1 New Handler

The requirements for a replacement `new_handler` are specified by [new.handler], paragraph 2:

> Required behavior: A `new_handler` shall perform one of the following:
>
> - make more storage available for allocation and then return;
> - throw an exception of type `bad_alloc` or a class derived from `bad_alloc`;
> - terminate execution of the program without returning to the caller;

10.7.1.2 Terminate Handler

The requirements for a replacement `terminate_handler` are specified by [terminate.handler], paragraph 2:

> Required behavior: A `terminate_handler` shall terminate execution of the program without returning to the caller.

10.7.1.3 Unexpected Handler

The requirements for a replacement `unexpected_handler` are specified by [unexpected.handler], paragraph 2.

> Required behavior: An `unexpected_handler` shall not return. See also 15.5.2.

`unexpected_handler` is a deprecated feature of C++.

10.7.2 Noncompliant Code Example

In this noncompliant code example, a replacement `new_handler` is written to attempt to release salvageable resources when the dynamic memory manager runs out of memory. However, this example does not take into account the situation in which all salvageable resources have been recovered and there is still insufficient memory to satisfy the allocation request. Instead of terminating the replacement handler with an exception of type `std::bad_alloc` or terminating the execution of the program without returning to the caller, the replacement handler returns as normal. Under low memory conditions, an infinite loop will occur with the default implementation of `::operator new()`. Because such conditions are rare in practice, it is likely for this bug to go undiscovered under typical testing scenarios.

```
#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);

  // ...
}
```

10.7.3 Compliant Solution

In this compliant solution, `custom_new_handler()` uses the return value from `reclaim_resources()`. If it returns `0`, then there will be insufficient memory for `operator new` to succeed. Hence, an exception of type `std::bad_alloc` is thrown, meeting the requirements for the replacement handler.

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

10.7.4 Risk Assessment

Failing to meet the required behavior for a replacement handler results in undefined behavior.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
OOP56-CPP	Low	Probable	High	**P2**	**L3**

10.7.5 Related Guidelines

SEI CERT C++ Coding Standard	MEM55-CPP. Honor replacement dynamic storage management requirements

10.7.6 Bibliography

[ISO/IEC 14882-2014]	Subclause 17.6.4.8, “Other Functions” Subclause 18.6.2.3, “Type `new_handler`” Subclause 18.8.3.1, “Type `terminate_handler`” Subclause D.11.1, “Type `unexpected_handler`”

10.8 OOP57-CPP. Prefer special member functions and overloaded operators to C Standard Library functions

Several C standard library functions perform bytewise operations on objects. For instance, `std::memcmp()` compares the bytes comprising the object representation of two objects, and `std::memcpy()` copies the bytes comprising an object representation into a destination buffer. However, for some object types, it results in undefined or abnormal program behavior.

The C++ Standard, [class], paragraph 6 [ISO/IEC 14882-2014], states the following:

> A *trivially copyable class* is a class that:
>
> - has no non-trivial copy constructors,
> - has no non-trivial move constructors,
> - has no non-trivial copy assignment operators,
> - has no non-trivial move assignment operators, and
> - has a trivial destructor.
>
> A *trivial class* is a class that has a default constructor, has no non-trivial default constructors, and is trivially copyable. [Note: In particular, a trivially copyable or trivial class does not have virtual functions or virtual base classes. — end note]

Additionally, the C++ Standard, [class], paragraph 7, states the following:

> A *standard-layout class* is a class that:
>
> - has no non-static data members of type non-standard-layout class (or array of such types) or reference,
> - has no virtual functions and no virtual base classes,
> - has the same access control for all non-static data members,
> - has no non-standard-layout base classes,
> - either has no non-static data members in the most derived class and at most one base class with non-static data members, or has no base classes with non-static data members, and
> - has no base classes of the same type as the first non-static data member.

Do not use `std::memset()` to initialize an object of nontrivial class type as it may not properly initialize the value representation of the object. Do not use `std::memcpy()` (or related bytewise copy functions) to initialize a copy of an object of nontrivial class type, as it may not properly initialize the value representation of the copy. Do not use `std::memcmp()` (or related bytewise comparison functions) to compare objects of nonstandard-layout class type, as it may not properly compare the value representations of the objects. In all cases, it is best to prefer the alternatives.

C Standard Library Function	C++ Equivalent Functionality
`std::memset()`	Class constructor

`std::memcpy()`
`std::memmove()`
`std::strcpy()`
	Class copy constructor or `operator=()`

`std::memcmp()`
`std::strcmp()`
	`operator<()`, `operator>()`, `operator==()`, or `operator!=()`

10.8.1 Noncompliant Code Example

In this noncompliant code example, a nontrivial class object is initialized by calling its default constructor but is later reinitialized to its default state using `std::memset()`, which does not properly reinitialize the object. Improper reinitialization leads to class invariants not holding in later uses of the object.

```
#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;
}
```

The above noncompliant code example is compliant with EXP62-CPP. Do not access the bits of an object representation that are not part of the object's value representation because all of the bits in the value representation are also used in the object representation of `C`.

10.8.2 Compliant Solution

In this compliant solution, the call to `std::memset()` is replaced with a default-initialized copy-and-swap operation called `clear()`. This operation ensures that the object is initialized to its default state properly, and it behaves properly for object types that have optimized assignment operators that fail to clear all data members of the object being assigned into.

```
#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;
}
```

10.8.3 Noncompliant Code Example

In this noncompliant code example, `std::memcpy()` is used to create a copy of an object of nontrivial type `C`. However, because each object instance attempts to delete the `int *` in `C::~C()`, double-free vulnerabilities may occur because the same pointer value will be copied into `c2`.

```
#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));
}
```

10.8.4 Compliant Solution

In this compliant solution, `C` defines an assignment operator that is used instead of calling `std::memcpy()`.

```
class C {
  int *i;

public:
  C() : i(nullptr) {}
  ~C() { delete i; }

  void set(int val) {
    if (i) { delete i; }
    i = new int{val};
  }

  C &operator=(const C &rhs) noexcept(false) {
    if (this != &rhs) {
      int *o = nullptr;
      if (rhs.i) {
        o = new int;
        *o = *rhs.i;
      }
      // Does not modify this unless allocation succeeds.
      delete i;
      i = o;
    }
    return *this;
  }

  // ...
};

void f(C &c1) {
  C c2 = c1;
}
```

10.8.5 Noncompliant Code Example

In this noncompliant code example, `std::memcmp()` is used to compared two objects of nonstandard-layout type. Because `std::memcmp()` performs a bytewise comparison of the object representations, if the implementation uses a vtable pointer as part of the object representation, it will compare vtable pointers. If the dynamic type of either `c1` or `c2` is a derived class of type `C`, the comparison may fail despite the value representation of either object.

```
#include <cstring>

class C {
  int i;

public:
  virtual void f();

  // ...
};

void f(C &c1, C &c2) {
  if (!std::memcmp(&c1, &c2, sizeof(C))) {
    // ...
  }
}
```

Because a vtable is not part of an object's value representation, comparing it with `std::memcmp()` also violates EXP62-CPP. Do not access the bits of an object representation that are not part of the object's value representation.

10.8.6 Compliant Solution

In this compliant solution, `C` defines an equality operator that is used instead of calling `std::memcmp()`. This solution ensures that only the value representation of the objects is considered when performing the comparison.

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

10.8.7 Risk Assessment

Most violations of this rule will result in abnormal program behavior. However, overwriting implementation details of the object representation can lead to code execution vulnerabilities.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
OOP57-CPP	High	Probable	High	**P6**	**L2**

10.8.8 Related Guidelines

SEI CERT C++ Coding Standard	EXP62-CPP. Do not access the bits of an object representation that are not part of the object's value representation

10.8.9 Bibliography

[ISO/IEC 14882-2014]	Subclause 3.9, "Types" Subclause 3.10, "Lvalues and Rvalues" Clause 9, "Classes"

10.9 OOP58-CPP. Copy operations must not mutate the source object

Copy operations (copy constructors and copy assignment operators) are expected to copy the salient properties of a source object into the destination object, with the resulting object being a "copy" of the original. What is considered to be a salient property of the type is type-dependent, but for types that expose comparison or equality operators, includes any properties used for those comparison operations. This expectation leads to assumptions in code that a copy operation results in a destination object with a value representation that is equivalent to the source object value representation. Violation of this basic assumption can lead to unexpected behavior.

Ideally, the copy operator should have an idiomatic signature. For copy constructors, that is `T(const T&);` and for copy assignment operators, that is `T& operator=(const T&);`. Copy constructors and copy assignment operators that do not use an idiomatic signature do not meet the requirements of the `CopyConstructible` or `CopyAssignable` concept, respectively. This precludes the type from being used with common standard library functionality [ISO/IEC 14882-2014].

When implementing a copy operator, do not mutate any externally observable members of the source object operand or globally accessible information. Externally observable members include, but are not limited to, members that participate in comparison or equality operations, members whose values are exposed via public APIs, and global variables.

Before C++11, a copy operation that mutated the source operand was the only way to provide move-like semantics. However, the language did not provide a way to enforce that this operation only occurred when the source operand was at the end of its lifetime, which led to fragile APIs like `std::auto_ptr`. In C++11 and later, such a situation is a good candidate for a move operation instead of a copy operation.

10.9.1.1 `auto_ptr`

For example, in C++03, `std::auto_ptr` had the following copy operation signatures [ISO/IEC 14882-2003]:

Copy constructor	`auto_ptr(auto_ptr &A);`
Copy assignment	`auto_ptr& operator=(auto_ptr &A);`

Both copy construction and copy assignment would mutate the source argument, `A`, by effectively calling `this->reset(A.release())`. However, this invalidated assumptions made by standard library algorithms such as `std::sort()`, which may need to make a copy of an object for later comparisons [Hinnant 2005]. Consider the following implementation of `std::sort()` that implements the quick sort algorithm.

```
// ...
value_type pivot_element = *mid_point;
// ...
```

At this point, the sorting algorithm assumes that `pivot_element` and `*mid_point` have equivalent value representations and will compare equal. However, for `std::auto_ptr`, this is not the case because `*mid_point` has been mutated and results in unexpected behavior.

In C++11, the `std::unique_ptr` smart pointer class was introduced as a replacement for `std::auto_ptr` to better specify the ownership semantics of pointer objects. Rather than mutate the source argument in a copy operation, `std::unique_ptr` explicitly deletes the copy constructor and copy assignment operator, and instead uses a move constructor and move assignment operator. Subsequently, `std::auto_ptr` was deprecated in C++11.

10.9.2 Noncompliant Code Example

In this noncompliant code example, the copy operations for `A` mutate the source operand by resetting its member variable `m` to `0`. When `std::fill()` is called, the first element copied will have the original value of `obj.m`, `12`, at which point `obj.m` is set to `0`. The subsequent nine copies will all retain the value `0`.

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

class A {
  mutable int m;

public:
  A() : m(0) {}
  explicit A(int m) : m(m) {}

  A(const A &other) : m(other.m) {
    other.m = 0;
  }

  A& operator=(const A &other) {
    if (&other != this) {
      m = other.m;
      other.m = 0;
    }
    return *this;
  }

  int get_m() const { return m; }
};

void f() {
  std::vector<A> v{10};
  A obj(12);
  std::fill(v.begin(), v.end(), obj);
}
```

10.9.3 Compliant Solution

In this compliant solution, the copy operations for A no longer mutate the source operand, ensuring that the vector contains equivalent copies of `obj`. Instead, A has been given move operations that perform the mutation when it is safe to do so.

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

class A {
  int m;

public:
  A() : m(0) {}
  explicit A(int m) : m(m) {}

  A(const A &other) : m(other.m) {}
  A(A &&other) : m(other.m) { other.m = 0; }

  A& operator=(const A &other) {
    if (&other != this) {
      m = other.m;
    }
    return *this;
  }

  A& operator=(A &&other) {
    m = other.m;
    other.m = 0;
    return *this;
  }

  int get_m() const { return m; }
};

void f() {
  std::vector<A> v{10};
  A obj(12);
  std::fill(v.begin(), v.end(), obj);
}
```

10.9.4 Risk Assessment

Copy operations that mutate the source operand or global state can lead to unexpected program behavior. Using such a type in a Standard Template Library container or algorithm can also lead to undefined behavior.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
OOP58-CPP	Low	Likely	Low	**P9**	**L2**

10.9.5 Related Guidelines

SEI CERT C++ Coding Standard	OOP54-CPP. Gracefully handle self-copy assignment

10.9.6 Bibliography

[ISO/IEC 14882-2014]	Subclause 12.8, “Copying and Moving Class Objects” Table 21, “CopyConstructible Requirements” Table 23, “CopyAssignable Requirements”
[ISO/IEC 14882-2003]	
[Hinnant 2005]	“Rvalue Reference Recommendations for Chapter 20”

11 Concurrency (CON)

11.1 CON50-CPP. Do not destroy a mutex while it is locked

Mutex objects are used to protect shared data from being concurrently accessed. If a mutex object is destroyed while a thread is blocked waiting for the lock, critical sections and shared data are no longer protected.

The C++ Standard, [thread.mutex.class], paragraph 5 [ISO/IEC 14882-2014], states the following:

> The behavior of a program is undefined if it destroys a `mutex` object owned by any thread or a thread terminates while owning a `mutex` object.

Similar wording exists for `std::recursive_mutex`, `std::timed_mutex`, `std::recursive_timed_mutex`, and `std::shared_timed_mutex`. These statements imply that destroying a mutex object while a thread is waiting on it is undefined behavior.

11.1.1 Noncompliant Code Example

This noncompliant code example creates several threads that each invoke the `do_work()` function, passing a unique number as an ID.

Unfortunately, this code contains a race condition, allowing the mutex to be destroyed while it is still owned, because `start_threads()` may invoke the mutex's destructor before all of the threads have exited.

```
#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);
  }
}
```

11.1.2 Compliant Solution

This compliant solution eliminates the race condition by extending the lifetime of the mutex.

```
#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);
  }
}
```

11.1.3 Compliant Solution

This compliant solution eliminates the race condition by joining the threads before the mutex's destructor is invoked.

```
#include <mutex>
#include <thread>

const size_t maxThreads = 10;

void do_work(size_t i, std::mutex *pm) {
  std::lock_guard<std::mutex> lk(*pm);

  // Access data protected by the lock.
}
void run_threads() {
  std::thread threads[maxThreads];
  std::mutex m;

  for (size_t i = 0; i < maxThreads; ++i) {
    threads[i] = std::thread(do_work, i, &m);
  }

  for (size_t i = 0; i < maxThreads; ++i) {
    threads[i].join();
  }
}
```

11.1.4 Risk Assessment

Destroying a mutex while it is locked may result in invalid control flow and data corruption.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
CON50-CPP	Medium	Probable	High	**P4**	**L3**

11.1.5 Related Guidelines

MITRE CWE	CWE-667, Improper Locking
SEI CERT C Coding Standard	CON31-C. Do not destroy a mutex while it is locked

11.1.6 Bibliography

[ISO/IEC 14882-2014]	Subclause 30.4.1, "Mutex Requirements"

11.2 CON51-CPP. Ensure actively held locks are released on exceptional conditions

Mutexes that are used to protect accesses to shared data may be locked using the `lock()` member function and unlocked using the `unlock()` member function. If an exception occurs between the call to `lock()` and the call to `unlock()`, and the exception changes control flow such that `unlock()` is not called, the mutex will be left in the locked state and no critical sections protected by that mutex will be allowed to execute. This is likely to lead to deadlock.

The throwing of an exception must not allow a mutex to remain locked indefinitely. If a mutex was locked and an exception occurs within the critical section protected by that mutex, the mutex must be unlocked as part of exception handling before rethrowing the exception or continuing execution unless subsequent control flow will unlock the mutex.

C++ supplies the lock classes `lock_guard`, `unique_lock`, and `shared_lock`, which can be initialized with a mutex. In its constructor, the lock object locks the mutex, and in its destructor, it unlocks the mutex. The `lock_guard` class provides a simple RAII wrapper around a mutex. The `unique_lock` and `shared_lock` classes also use RAII and provide additional functionality, such as manual control over the locking strategy. The `unique_lock` class prevents the lock from being copied, although it allows the lock ownership to be moved to another lock. The `shared_lock` class allows the mutex to be shared by several locks. For all three classes, if an exception occurs and takes control flow out of the scope of the lock, the destructor will unlock the mutex and the program can continue working normally. These lock objects are the preferred way to ensure that a mutex is properly released when an exception is thrown.

11.2.1 Noncompliant Code Example

This noncompliant code example manipulates shared data and protects the critical section by locking the mutex. When it is finished, it unlocks the mutex. However, if an exception occurs while manipulating the shared data, the mutex will remain locked.

```
#include <mutex>

void manipulate_shared_data(std::mutex &pm) {
  pm.lock();

  // Perform work on shared data.

  pm.unlock();
}
```

11.2.2 Compliant Solution (Manual Unlock)

This compliant solution catches any exceptions thrown when performing work on the shared data and unlocks the mutex before rethrowing the exception.

```
#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
}
```

11.2.3 Compliant Solution (Lock Object)

This compliant solution uses a `lock_guard` object to ensure that the mutex will be unlocked, even if an exception occurs, without relying on exception handling machinery and manual resource management.

```
#include <mutex>

void manipulate_shared_data(std::mutex &pm) {
  std::lock_guard<std::mutex> lk(pm);

  // Perform work on shared data.
}
```

11.2.4 Risk Assessment

If an exception occurs while a mutex is locked, deadlock may result.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
CON51-CPP	Low	Probable	Low	**P6**	**L2**

11.2.5 Related Guidelines

This rule is a subset of ERR56-CPP. Guarantee exception safety.

MITRE CWE	CWE-667, Improper Locking
SEI CERT Oracle Coding Standard for Java	LCK08-J. Ensure actively held locks are released on exceptional conditions

11.2.6 Bibliography

[ISO/IEC 14882-2014]	Subclause 30.4.2, “Locks”

11.3 CON52-CPP. Prevent data races when accessing bit-fields from multiple threads

When accessing a bit-field, a thread may inadvertently access a separate bit-field in adjacent memory. This is because compilers are required to store multiple adjacent bit-fields in one storage unit whenever they fit. Consequently, data races may exist not just on a bit-field accessed by multiple threads but also on other bit-fields sharing the same byte or word. The problem is difficult to diagnose because it may not be obvious that the same memory location is being modified by multiple threads.

One approach for preventing data races in concurrent programming is to use a mutex. When properly observed by all threads, a mutex can provide safe and secure access to a shared object. However, mutexes provide no guarantees with regard to other objects that might be accessed when the mutex is not controlled by the accessing thread. Unfortunately, there is no portable way to determine which adjacent bit-fields may be stored along with the desired bit-field.

Another approach is to insert a non-bit-field member between any two bit-fields to ensure that each bit-field is the only one accessed within its storage unit. This technique effectively guarantees that no two bit-fields are accessed simultaneously.

11.3.1 Noncompliant Code Example (bit-field)

Adjacent bit-fields may be stored in a single memory location. Consequently, modifying adjacent bit-fields in different threads is undefined behavior, as shown in this noncompliant code example.

```
struct MultiThreadedFlags {
  unsigned int flag1 : 2;
  unsigned int flag2 : 2;
};

MultiThreadedFlags flags;

void thread1() {
  flags.flag1 = 1;
}

void thread2() {
  flags.flag2 = 2;
}
```

For example, the following instruction sequence is possible.

```
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
```

11.3.2 Compliant Solution (bit-field, C++11 and later, mutex)

This compliant solution protects all accesses of the flags with a mutex, thereby preventing any data races.

```
#include <mutex>

struct MultiThreadedFlags {
  unsigned int flag1 : 2;
  unsigned int flag2 : 2;
};

struct MtfMutex {
  MultiThreadedFlags s;
  std::mutex mutex;
};

MtfMutex flags;

void thread1() {
  std::lock_guard<std::mutex> lk(flags.mutex);
  flags.s.flag1 = 1;
}

void thread2() {
  std::lock_guard<std::mutex> lk(flags.mutex);
  flags.s.flag2 = 2;
}
```

11.3.3 Compliant Solution (C++11)

In this compliant solution, two threads simultaneously modify two distinct non-bit-field members of a structure. Because the members occupy different bytes in memory, no concurrency protection is required.

```
struct MultiThreadedFlags {
  unsigned char flag1;
  unsigned char flag2;
};

MultiThreadedFlags flags;

void thread1() {
  flags.flag1 = 1;
}

void thread2() {
  flags.flag2 = 2;
}
```

Unlike earlier versions of the standard, C++11 and later explicitly define a memory location and provide the following note in [intro.memory] paragraph 4 [ISO/IEC 14882-2014]:

> [*Note:* Thus a bit-field and an adjacent non-bit-field are in separate memory locations, and therefore can be concurrently updated by two threads of execution without interference. The same applies to two bit-fields, if one is declared inside a nested struct declaration and the other is not, or if the two are separated by a zero-length bit-field declaration, or if they are separated by a non-bit-field declaration. It is not safe to concurrently update two bit-fields in the same struct if all fields between them are also bit-fields of non-zero width. – *end note*]

It is almost certain that `flag1` and `flag2` are stored in the same word. Using a compiler that conforms to earlier versions of the standard, if both assignments occur on a thread-scheduling interleaving that ends with both stores occurring after one another, it is possible that only one of the flags will be set as intended, and the other flag will contain its previous value because both members are represented by the same word, which is the smallest unit the processor can work on. Before the changes made to the C++ Standard for C++11, there were no guarantees that these flags could be modified concurrently.

11.3.4 Risk Assessment

Although the race window is narrow, an assignment or an expression can evaluate improperly because of misinterpreted data resulting in a corrupted running state or unintended information disclosure.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
CON52-CPP	Medium	Probable	Medium	**P8**	**L2**

11.3.5 Related Guidelines

SEI CERT C Coding Standard	CON32-C. Prevent data races when accessing bit-fields from multiple threads

11.3.6 Bibliography

[ISO/IEC 14882-2014]	Subclause 1.7, “The C++ memory model”

11.4 CON53-CPP. Avoid deadlock by locking in a predefined order

Mutexes are used to prevent multiple threads from causing a data race by accessing the same shared resource at the same time. Sometimes, when locking mutexes, multiple threads hold each other's lock, and the program consequently deadlocks. Four conditions are required for deadlock to occur:

- mutual exclusion (At least one nonshareable resource must be held.),
- hold and wait (A thread must hold a resource while awaiting availability of another resource.),
- no preemption (Resources cannot be taken away from a thread while they are in-use.), and
- circular wait (A thread must await a resource held by another thread which is, in turn, awaiting a resource held by the first thread.).

Deadlock needs all four conditions, so preventing deadlock requires preventing any one of the four conditions. One simple solution is to lock the mutexes in a predefined order, which prevents circular wait.

11.4.1 Noncompliant Code Example

The behavior of this noncompliant code example depends on the runtime environment and the platform's scheduler. The program is susceptible to deadlock if thread `thr1` attempts to lock `ba2`'s mutex at the same time thread `thr2` attempts to lock `ba1`'s mutex in the `deposit()` function.

```
#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();
}
```

11.4.2 Compliant Solution (Manual Ordering)

This compliant solution eliminates the circular wait condition by establishing a predefined order for locking in the `deposit()` function. Each thread will lock on the basis of the `BankAccount` ID, which is set when the `BankAccount` object is initialized.

```
#include <atomic>
#include <mutex>
#include <thread>

class BankAccount {
  static std::atomic<unsigned int> globalId;
  const unsigned int id;
  int balance;
public:
  std::mutex balanceMutex;
  BankAccount() = delete;
  explicit BankAccount(int initialAmount) :
      id(globalId++), balance(initialAmount) {}
  unsigned int get_id() const { return id; }
  int get_balance() const { return balance; }
  void set_balance(int amount) { balance = amount; }
};

std::atomic<unsigned int> BankAccount::globalId(1);

int deposit(BankAccount *from, BankAccount *to, int amount) {
  std::mutex *first;
  std::mutex *second;

  if (from->get_id() == to->get_id()) {
    return -1; // Indicate error
  }

  // Ensure proper ordering for locking.
  if (from->get_id() < to->get_id()) {
    first = &from->balanceMutex;
    second = &to->balanceMutex;
  } else {
    first = &to->balanceMutex;
    second = &from->balanceMutex;
  }
  std::lock_guard<std::mutex> firstLock(*first);
  std::lock_guard<std::mutex> secondLock(*second);

  // Check for enough balance to transfer.
  if (from->get_balance() >= amount) {
    from->set_balance(from->get_balance() – amount);
    to->set_balance(to->get_balance() + amount);
    return 0;
  }
  return -1;
}

void f(BankAccount *ba1, BankAccount *ba2) {
  // Perform the deposits.
  std::thread thr1(deposit, ba1, ba2, 100);
  std::thread thr2(deposit, ba2, ba1, 100);
  thr1.join();
  thr2.join();
}
```

11.4.3 Compliant Solution (`std::lock()`)

This compliant solution uses Standard Template Library facilities to ensure that deadlock does not occur due to circular wait conditions. The `std::lock()` function takes a variable number of lockable objects and attempts to lock them such that deadlock does not occur [ISO/IEC 14882-2014]. In typical implementations, this is done by using a combination of `lock()`, `try_lock()`, and `unlock()` to attempt to lock the object and backing off if the lock is not acquired, which may have worse performance than a solution that locks in predefined order explicitly.

```
#include <mutex>
#include <thread>

class BankAccount {
  int balance;
public:
  std::mutex balanceMutex;
  BankAccount() = delete;
  explicit BankAccount(int initialAmount) :
    balance(initialAmount) {}
  int get_balance() const { return balance; }
  void set_balance(int amount) { balance = amount; }
};

int deposit(BankAccount *from, BankAccount *to, int amount) {
  // Create lock objects but defer locking them until later.
  std::unique_lock<std::mutex> lk1(
    from->balanceMutex, std::defer_lock);
  std::unique_lock<std::mutex> lk2(
    to->balanceMutex, std::defer_lock);

  // Lock both of the lock objects simultaneously.
  std::lock(lk1, lk2);

  if (from->get_balance() >= amount) {
    from->set_balance(from->get_balance() – amount);
    to->set_balance(to->get_balance() + amount);
    return 0;
  }
  return -1;
}

void f(BankAccount *ba1, BankAccount *ba2) {
  // Perform the deposits.
  std::thread thr1(deposit, ba1, ba2, 100);
  std::thread thr2(deposit, ba2, ba1, 100);
  thr1.join();
  thr2.join();
}
```

11.4.4 Risk Assessment

Deadlock prevents multiple threads from progressing, halting program execution. A denial-of-service attack is possible if the attacker can create the conditions for deadlock.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
CON53-CPP	Low	Probable	Medium	**P4**	**L3**

11.4.5 Related Guidelines

CERT Oracle Secure Coding Standard for Java	LCK07-J. Avoid deadlock by requesting and releasing locks in the same order
SEI CERT C Coding Standard	CON35-C. Avoid deadlock by locking in a predefined order
MITRE CWE	CWE-764, Multiple Locks of a Critical Resource

11.4.6 Bibliography

[ISO/IEC 14882-2014]	Subclause 30.4, “Mutual Exclusion” Subclause 30.4.3, “Generic Locking Algorithms”

11.5 CON54-CPP. Wrap functions that can spuriously wake up in a loop

The `wait()`, `wait_for()`, and `wait_until()` member functions of the `std::condition_variable` class temporarily cede possession of a mutex so that other threads that may be requesting the mutex can proceed. These functions must always be called from code that is protected by locking a mutex. The waiting thread resumes execution only after it has been notified, generally as the result of the invocation of the `notify_one()` or `notify_all()` member functions invoked by another thread.

The `wait()` function must be invoked from a loop that checks whether a condition predicate holds. A condition predicate is an expression constructed from the variables of a function that must be true for a thread to be allowed to continue execution. The thread pauses execution via `wait()`, `wait_for()`, `wait_until()`, or some other mechanism, and is resumed later, presumably when the condition predicate is true and the thread is notified.

```
#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.
}
```

The notification mechanism notifies the waiting thread and allows it to check its condition predicate. The invocation of `notify_all()` in another thread cannot precisely determine which waiting thread will be resumed. Condition predicate statements allow notified threads to determine whether they should resume upon receiving the notification.

11.5.1 Noncompliant Code Example

This noncompliant code example monitors a linked list and assigns one thread to consume list elements when the list is nonempty.

This thread pauses execution using `wait()` and resumes when notified, presumably when the list has elements to be consumed. It is possible for the thread to be notified even if the list is still empty, perhaps because the notifying thread used `notify_all()`, which notifies all threads. Notification using `notify_all()` is frequently preferred over using `notify_one()`. (See CON55-CPP. Preserve thread safety and liveness when using condition variables for more information.)

A condition predicate is typically the negation of the condition expression in the loop. In this noncompliant code example, the condition predicate for removing an element from a linked list is `(list->next != nullptr)`, whereas the condition expression for the `while` loop condition is `(list->next == nullptr)`.

This noncompliant code example nests the call to `wait()` inside an `if` block and consequently fails to check the condition predicate after the notification is received. If the notification was spurious or malicious, the thread would wake up prematurely.

```
#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.
}
```

11.5.2 Compliant Solution (Explicit loop with predicate)

This compliant solution calls the `wait()` member function from within a `while` loop to check the condition both before and after the call to `wait()`.

```
#include <condition_variable>
#include <mutex>

struct Node {
  void *node;
  struct Node *next;
};

static Node list;
static std::mutex m;
static std::condition_variable condition;

void consume_list_element() {
  std::unique_lock<std::mutex> lk(m);

  while (list.next == nullptr) {
    condition.wait(lk);
  }

  // Proceed when condition holds.
}
```

11.5.3 Compliant Solution (Implicit loop with lambda predicate)

The `std::condition_variable::wait()` function has an overloaded form that accepts a function object representing the predicate. This form of `wait()` behaves as if it were implemented as `while (!pred()) wait(lock);`. This compliant solution uses a lambda as a predicate and passes it to the `wait()` function. The predicate is expected to return true when it is safe to proceed, which reverses the predicate logic from the compliant solution using an explicit loop predicate.

```
#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.
}
```

11.5.4 Risk Assessment

Failure to enclose calls to the `wait()`, `wait_for()`, or `wait_until()` member functions inside a `while` loop can lead to indefinite blocking and denial of service (DoS).

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
CON54-CPP	Low	Unlikely	Medium	**P2**	**L3**

11.5.5 Related Guidelines

CERT Oracle Secure Coding Standard for Java	THI03-J. Always invoke wait() and await() methods inside a loop
SEI CERT C Coding Standard	CON36-C. Wrap functions that can spuriously wake up in a loop
SEI CERT C++ Coding Standard	CON55-CPP. Preserve thread safety and liveness when using condition variables

11.5.6 Bibliography

[ISO/IEC 9899:2011]	7.17.7.4, "The `atomic_compare_exchange` Generic Functions"
[Lea 2000]	1.3.2, "Liveness" 3.2.2, "Monitor Mechanics"

11.6 CON55-CPP. Preserve thread safety and liveness when using condition variables

Both thread safety and liveness are concerns when using condition variables. The *thread-safety* property requires that all objects maintain consistent states in a multithreaded environment [Lea 2000]. The *liveness* property requires that every operation or function invocation execute to completion without interruption; for example, there is no deadlock.

Condition variables must be used inside a `while` loop. (See CON54-CPP. Wrap functions that can spuriously wake up in a loop for more information.) To guarantee liveness, programs must test the `while` loop condition before invoking the `condition_variable::wait()` member function. This early test checks whether another thread has already satisfied the condition predicate and has sent a notification. Invoking `wait()` after the notification has been sent results in indefinite blocking.

To guarantee thread safety, programs must test the `while` loop condition after returning from `wait()`. When a given thread invokes `wait()`, it will attempt to block until its condition variable is signaled by a call to `condition_variable::notify_all()` or to `condition_variable::notify_one()`.

The `notify_one()` member function unblocks one of the threads that are blocked on the specified condition variable at the time of the call. If multiple threads are waiting on the same condition variable, the scheduler can select any of those threads to be awakened (assuming that all threads have the same priority level).

The `notify_all()` member function unblocks all of the threads that are blocked on the specified condition variable at the time of the call. The order in which threads execute following a call to `notify_all()` is unspecified. Consequently, an unrelated thread could start executing, discover that its condition predicate is satisfied, and resume execution even though it was supposed to remain dormant.

For these reasons, threads must check the condition predicate after the `wait()` function returns. A `while` loop is the best choice for checking the condition predicate both before and after invoking `wait()`.

The use of `notify_one()` is safe if each thread uses a unique condition variable. If multiple threads share a condition variable, the use of `notify_one()` is safe only if the following conditions are met:

- All threads must perform the same set of operations after waking up, which means that any thread can be selected to wake up and resume for a single invocation of `notify_one()`.
- Only one thread is required to wake upon receiving the signal.

The `notify_all()` function can be used to unblock all of the threads that are blocked on the specified condition variable if the use of `notify_one()` is unsafe.

11.6.1 Noncompliant Code Example (`notify_one()`)

This noncompliant code example uses five threads that are intended to execute sequentially according to the step level assigned to each thread when it is created (serialized processing). The `currentStep` variable holds the current step level and is incremented when the respective thread completes. Finally, another thread is signaled so that the next step can be executed. Each thread waits until its step level is ready, and the `wait()` call is wrapped inside a `while` loop, in compliance with CON54-CPP. Wrap functions that can spuriously wake up in a loop.

```
#include <condition_variable>
#include <iostream>
#include <mutex>
#include <thread>

std::mutex mutex;
std::condition_variable cond;

void run_step(size_t myStep) {
  static size_t currentStep = 0;
  std::unique_lock<std::mutex> lk(mutex);

  std::cout << "Thread " << myStep << " has the lock" << std::endl;

  while (currentStep != myStep) {
    std::cout << "Thread " << myStep << " is sleeping..."
              << std::endl;
    cond.wait(lk);
    std::cout << "Thread " << myStep << " woke up" << std::endl;
  }

  // Do processing...
  std::cout << "Thread " << myStep << " is processing..."
            << std::endl;
  currentStep++;

  // Signal awaiting task.
  cond.notify_one();

  std::cout << "Thread " << myStep << " is exiting..."
            << std::endl;
}

int main() {
  constexpr size_t numThreads = 5;
  std::thread threads[numThreads];

  // Create threads.
  for (size_t i = 0; i < numThreads; ++i) {
    threads[i] = std::thread(run_step, i);
  }

  // Wait for all threads to complete.
  for (size_t i = numThreads; i != 0; --i) {
    threads[i - 1].join();
  }
}
```

In this example, all threads share a single condition variable. Each thread has its own distinct condition predicate because each thread requires `currentStep` to have a different value before proceeding. When the condition variable is signaled, any of the waiting threads can wake up. The following table illustrates a possible scenario in which the liveness property is violated. If, by chance, the notified thread is not the thread with the next step value, that thread will wait again. No additional notifications can occur, and eventually the pool of available threads will be exhausted.

Deadlock: Out-of-Sequence Step Value

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.

This noncompliant code example violates the liveness property.

11.6.2 Compliant Solution (`notify_all()`)

This compliant solution uses `notify_all()` to signal all waiting threads instead of a single random thread. Only the `run_step()` thread code from the noncompliant code example is modified.

```
#include <condition_variable>
#include <iostream>
#include <mutex>
#include <thread>

std::mutex mutex;
std::condition_variable cond;

void run_step(size_t myStep) {
  static size_t currentStep = 0;
  std::unique_lock<std::mutex> lk(mutex);

  std::cout << "Thread " << myStep << " has the lock" << std::endl;

  while (currentStep != myStep) {
    std::cout << "Thread " << myStep << " is sleeping..."
              << std::endl;
    cond.wait(lk);
    std::cout << "Thread " << myStep << " woke up" << std::endl;
  }

  // Do processing ...
  std::cout << "Thread " << myStep << " is processing..."
            << std::endl;
  currentStep++;

  // Signal ALL waiting tasks.
  cond.notify_all();

  std::cout << "Thread " << myStep << " is exiting..."
            << std::endl;
}

// ... main() unchanged ...
```

Awakening all threads guarantees the liveness property because each thread will execute its condition predicate test, and exactly one will succeed and continue execution.

11.6.3 Compliant Solution (Using `notify_one()` with a Unique Condition Variable per Thread)

Another compliant solution is to use a unique condition variable for each thread (all associated with the same mutex). In this case, `notify_one()` wakes up only the thread that is waiting on it. This solution is more efficient than using `notify_all()` because only the desired thread is awakened.

The condition predicate of the signaled thread must be true; otherwise, a deadlock will occur.

```
#include <condition_variable>
#include <iostream>
#include <mutex>
#include <thread>

constexpr size_t numThreads = 5;

std::mutex mutex;
std::condition_variable cond[numThreads];

void run_step(size_t myStep) {
  static size_t currentStep = 0;
  std::unique_lock<std::mutex> lk(mutex);

  std::cout << "Thread " << myStep << " has the lock" << std::endl;

  while (currentStep != myStep) {
    std::cout << "Thread " << myStep << " is sleeping..."
              << std::endl;
    cond[myStep].wait(lk);
    std::cout << "Thread " << myStep << " woke up" << std::endl;
  }

  // Do processing ...
  std::cout << "Thread " << myStep << " is processing..."
            << std::endl;
  currentStep++;

  // Signal next step thread.
  if ((myStep + 1) < numThreads) {
    cond[myStep + 1].notify_one();
  }

  std::cout << "Thread " << myStep << " is exiting..."
            << std::endl;
}

// ... main() unchanged ...
```

11.6.4 Risk Assessment

Failing to preserve the thread safety and liveness of a program when using condition variables can lead to indefinite blocking and denial of service (DoS).

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
CON55-CPP	Low	Unlikely	Medium	**P2**	**L3**

11.6.5 Related Guidelines

CERT Oracle Secure Coding Standard for Java	THI02-J. Notify all waiting threads rather than a single thread
SEI CERT C Coding Standard	CON38-C. Preserve thread safety and liveness when using condition variables
SEI CERT C++ Coding Standard	CON54-CPP. Wrap functions that can spuriously wake up in a loop

11.6.6 Bibliography

[IEEE Std 1003.1:2013]	XSH, System Interfaces, `pthread_cond_broadcast` XSH, System Interfaces, `pthread_cond_signal`
[Lea 2000]	

11.7 CON56-CPP. Do not speculatively lock a non-recursive mutex that is already owned by the calling thread

The C++ Standard Library supplies both recursive and non-recursive mutex classes used to protect critical sections. The recursive mutex classes (`std::recursive_mutex` and `std::recursive_timed_mutex`) differ from the non-recursive mutex classes (`std::mutex`, `std::timed_mutex`, and `std::shared_timed_mutex`) in that a recursive mutex may be locked recursively by the thread that currently owns the mutex. All mutex classes support the ability to speculatively lock the mutex through functions such as `try_lock()`, `try_lock_for()`, `try_lock_until()`, `try_lock_shared_for ()`, and `try_lock_shared_until()`. These speculative locking functions attempt to obtain ownership of the mutex for the calling thread, but will not block in the event the ownership cannot be obtained. Instead, they return a Boolean value specifying whether the ownership of the mutex was obtained or not.

The C++ Standard, [thread.mutex.requirements.mutex], paragraphs 14 and 15 [ISO/IEC 14882-2014], state the following:

> The expression `m.try_lock()` shall be well-formed and have the following semantics: Requires: If `m` is of type `std::mutex`, `std::timed_mutex`, or `std::shared_timed_mutex`, the calling thread does not own the mutex.

Further, [thread.timedmutex.class], paragraph 3, in part, states the following:

> The behavior of a program is undefined if:
>
> - a thread that owns a `timed_mutex` object calls `lock()`, `try_lock()`, `try_lock_for()`, or `try_lock_until()` on that object

Finally, [thread.sharedtimedmutex.class], paragraph 3, in part, states the following:

> The behavior of a program is undefined if:
>
> - a thread attempts to recursively gain any ownership of a `shared_timed_mutex`.

Thus, attempting to speculatively lock a non-recursive mutex object that is already owned by the calling thread is undefined behavior. Do not call `try_lock()`, `try_lock_for()`, `try_lock_until()`, `try_lock_shared_for()`, or `try_lock_shared_until()` on a non-recursive mutex object from a thread that already owns that mutex object.

11.7.1 Noncompliant Code Example

In this noncompliant code example, the mutex `m` is locked by the thread's initial entry point and is speculatively locked in the `do_work()` function from the same thread, resulting in undefined behavior because it is not a recursive mutex. With common implementations, this may result in deadlock.

```
#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();
}
```

11.7.2 Compliant Solution

This compliant solution removes the lock from the thread’s initial entry point, allowing the mutex to be speculatively locked, but not recursively.

```
#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();
}
```

11.7.3 Risk Assessment

Speculatively locking a non-recursive mutex in a recursive manner is undefined behavior that can lead to deadlock.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
CON56-CPP	Low	Unlikely	High	**P1**	**L3**

11.7.4 Related Guidelines

MITRE CWE	CWE-667, Improper Locking

11.7.5 Bibliography

[ISO/IEC 14882-2014]	Subclause 30.4.1, “Mutex Requirements”

12 Miscellaneous (MSC)

12.1 MSC50-CPP. Do not use std::rand() for generating pseudorandom numbers

Pseudorandom number generators use mathematical algorithms to produce a sequence of numbers with good statistical properties, but the numbers produced are not genuinely random.

The C Standard `rand()` function, exposed through the C++ standard library through `<cstdlib>` as `std::rand()`, makes no guarantees as to the quality of the random sequence produced. The numbers generated by some implementations of `std::rand()` have a comparatively short cycle, and the numbers can be predictable. Applications that have strong pseudorandom number requirements must use a generator that is known to be sufficient for their needs.

12.1.1 Noncompliant Code Example

The following noncompliant code generates an ID with a numeric part produced by calling the `rand()` function. The IDs produced are predictable and have limited randomness. Further, depending on the value of `RAND_MAX`, the resulting value can have modulo bias.

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

void f() {
  std::string id("ID"); // Holds the ID, starting with the
                        // characters "ID" followed by a
                        // random integer in the range [0-10000].
  id += std::to_string(std::rand() % 10000);
  // ...
}
```

12.1.2 Compliant Solution

The C++ standard library provides mechanisms for fine-grained control over pseudorandom number generation. It breaks random number generation into two parts: one is the algorithm responsible for providing random values (the engine), and the other is responsible for distribution of the random values via a density function (the distribution). The distribution object is not strictly required, but it works to ensure that values are properly distributed within a given range instead of improperly distributed due to bias issues. This compliant solution uses the Mersenne Twister algorithm as the engine for generating random values and a uniform distribution to negate the modulo bias from the noncompliant code example.

```
#include <random>
#include <string>

void f() {
  std::string id("ID"); // Holds the ID, starting with the
                        // characters "ID" followed by a random
                        // integer in the range [0-10000].
  std::uniform_int_distribution<int> distribution(0, 10000);
  std::random_device rd;
  std::mt19937 engine(rd());
  id += std::to_string(distribution(engine));
  // ...
}
```

This compliant solution also seeds the random number engine, in conformance with MSC51-CPP. Ensure your random number generator is properly seeded.

12.1.3 Risk Assessment

Using the `std::rand()` function could lead to predictable random numbers.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
MSC50-CPP	Medium	Unlikely	Low	**P6**	**L2**

12.1.4 Related Guidelines

SEI CERT C++ Coding Standard	MSC51-CPP. Ensure your random number generator is properly seeded
SEI CERT C Coding Standard	MSC30-C. Do not use the rand() function for generating pseudorandom numbers
CERT Oracle Secure Coding Standard for Java	MSC02-J. Generate strong random numbers
MITRE CWE	CWE-327, Use of a Broken or Risky Cryptographic Algorithm CWE-330, Use of Insufficiently Random Values

12.1.5 Bibliography

[ISO/IEC 9899:2011]	Subclause 7.22.2, “Pseudo-random Sequence Generation Functions”
[ISO/IEC 14882-2014]	Subclause 26.5, “Random Number Generation”

12.2 MSC51-CPP. Ensure your random number generator is properly seeded

A pseudorandom number generator (PRNG) is a deterministic algorithm capable of generating sequences of numbers that approximate the properties of random numbers. Each sequence is completely determined by the initial state of the PRNG and the algorithm for changing the state. Most PRNGs make it possible to set the initial state, also called the *seed state*. Setting the initial state is called *seeding* the PRNG.

Calling a PRNG in the same initial state, either without seeding it explicitly or by seeding it with a constant value, results in generating the same sequence of random numbers in different runs of the program. Consider a PRNG function that is seeded with some initial seed value and is consecutively called to produce a sequence of random numbers. If the PRNG is subsequently seeded with the same initial seed value, then it will generate the same sequence.

Consequently, after the first run of an improperly seeded PRNG, an attacker can predict the sequence of random numbers that will be generated in the future runs. Improperly seeding or failing to seed the PRNG can lead to vulnerabilities, especially in security protocols.

The solution is to ensure that a PRNG is always properly seeded with an initial seed value that will not be predictable or controllable by an attacker. A properly seeded PRNG will generate a different sequence of random numbers each time it is run.

Not all random number generators can be seeded. True random number generators that rely on hardware to produce completely unpredictable results do not need to be and cannot be seeded. Some high-quality PRNGs, such as the `/dev/random` device on some UNIX systems, also cannot be seeded. This rule applies only to algorithmic PRNGs that can be seeded.

12.2.1 Noncompliant Code Example

This noncompliant code example generates a sequence of 10 pseudorandom numbers using the Mersenne Twister engine. No matter how many times this code is executed, it always produces the same sequence because the default seed is used for the engine.

```
#include <random>
#include <iostream>

void f() {
  std::mt19937 engine;

  for (int i = 0; i < 10; ++i) {
    std::cout << engine() << ", ";
  }
}
```

The output of this example follows.

```
1st run: 3499211612, 581869302, 3890346734, 3586334585, 545404204,
4161255391, 3922919429, 949333985, 2715962298, 1323567403,

2nd run: 3499211612, 581869302, 3890346734, 3586334585, 545404204,
4161255391, 3922919429, 949333985, 2715962298, 1323567403,

...

nth run: 3499211612, 581869302, 3890346734, 3586334585, 545404204,
4161255391, 3922919429, 949333985, 2715962298, 1323567403,
```

12.2.2 Noncompliant Code Example

This noncompliant code example improves the previous noncompliant code example by seeding the random number generation engine with the current time. However, this approach is still unsuitable when an attacker can control the time at which the seeding is executed. Predictable seed values can result in exploits when the subverted PRNG is used.

```
#include <ctime>
#include <random>
#include <iostream>

void f() {
  std::time_t t;
  std::mt19937 engine(std::time(&t));

  for (int i = 0; i < 10; ++i) {
    std::cout << engine() << ", ";
  }
}
```

12.2.3 Compliant Solution

This compliant solution uses `std::random_device` to generate a random value for seeding the Mersenne Twister engine object. The values generated by `std::random_device` are nondeterministic random numbers when possible, relying on random number generation devices, such as `/dev/random`. When such a device is not available, `std::random_device` may employ a random number engine; however, the initial value generated should have sufficient randomness to serve as a seed value.

```
#include <random>
#include <iostream>

void f() {
  std::random_device dev;
  std::mt19937 engine(dev());

  for (int i = 0; i < 10; ++i) {
    std::cout << engine() << ", ";
  }
}
```

The output of this example follows.

```
1st run: 3921124303, 1253168518, 1183339582, 197772533, 83186419,
2599073270, 3238222340, 101548389, 296330365, 3335314032,

2nd run: 2392369099, 2509898672, 2135685437, 3733236524,
883966369, 2529945396, 764222328, 138530885, 4209173263,
1693483251,

3rd run: 914243768, 2191798381, 2961426773, 3791073717,
2222867426, 1092675429, 2202201605, 850375565, 3622398137,
422940882,

...
```

12.2.4 Risk Assessment

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
MSC51-CPP	Medium	Likely	Low	**P18**	**L1**

12.2.4.1 Related Vulnerabilities

Using a predictable seed value, such as the current time, result in numerous vulnerabilities, such as the one described by CVE-2008-1637.

12.2.5 Related Guidelines

SEI CERT C Coding Standard	MSC32-C. Properly seed pseudorandom number generators
MITRE CWE	CWE-327, Use of a Broken or Risky Cryptographic Algorithm CWE-330, Use of Insufficiently Random Values CWE-337, Predictable Seed in PRNG

12.2.6 Bibliography

[ISO/IEC 9899:2011]	Subclause 7.22.2, “Pseudo-random Sequence Generation Functions”
[ISO/IEC 14882-2014]	Subclause 26.5, “Random Number Generation”

12.3 MSC52-CPP. Value-returning functions must return a value from all exit paths

The C++ Standard, [stmt.return], paragraph 2 [ISO/IEC 14882-2014], states the following:

> Flowing off the end of a function is equivalent to a `return` with no value; this results in undefined behavior in a value-returning function.

A value-returning function must return a value from all code paths; otherwise, it will result in undefined behavior. This includes returning through less-common code paths, such as from a *function-try-block*, as explained in the C++ Standard, [except.handle], paragraph 15:

> Flowing off the end of a *function-try-block* is equivalent to a `return` with no value; this results in undefined behavior in a value-returning function (6.6.3).

12.3.1 Noncompliant Code Example

In this noncompliant code example, the programmer forgot to return the input value for positive input, so not all code paths return a value.

```
int absolute_value(int a) {
  if (a < 0) {
    return -a;
  }
}
```

12.3.2 Compliant Solution

In this compliant solution, all code paths now return a value.

```
int absolute_value(int a) {
  if (a < 0) {
    return -a;
  }
  return a;
}
```

12.3.3 Noncompliant Code Example

In this noncompliant code example, the *function-try-block* handler does not return a value, resulting in undefined behavior when an exception is thrown.

```
#include <vector>

std::size_t f(std::vector<int> &v, std::size_t s) try {
  v.resize(s);
  return s;
} catch (...) {
}
```

12.3.4 Compliant Solution

In this compliant solution, the exception handler of the *function-try-block* also returns a value.

```
#include <vector>

std::size_t f(std::vector<int> &v, std::size_t s) try {
  v.resize(s);
  return s;
} catch (...) {
  return 0;
}
```

12.3.5 Exceptions

MSC54-CPP-EX1: Flowing off the end of the `main()` function is equivalent to a `return 0;` statement, according to the C++ Standard, [basic.start.main], paragraph 5 [ISO/IEC 14882-2014]. Thus, flowing off the end of the `main()` function does not result in undefined behavior.

MSC54-CPP-EX2: It is permissible for a control path to not return a value if that code path is never expected to be taken and a function marked `[[noreturn]]` is called as part of that code path or if an exception is thrown, as is illustrated in the following code example.

```
#include <cstdlib>
#include <iostream>
[[noreturn]] void unreachable(const char *msg) {
  std::cout << "Unreachable code reached: " << msg << std::endl;
  std::exit(1);
}

enum E {
  One,
  Two,
  Three
};

int f(E e) {
  switch (e) {
  case One: return 1;
  case Two: return 2;
  case Three: return 3;
  }
  unreachable("Can never get here");
}
```

12.3.6 Risk Assessment

Failing to return a value from a code path in a value-returning function results in undefined behavior that might be exploited to cause data integrity violations.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
MSC52-CPP	Medium	Probable	Medium	**P8**	**L2**

12.3.7 Bibliography

[ISO/IEC 14882-2014]	Subclause 3.6.1, "Main Function" Subclause 6.6.3, "The `return` Statement" Subclause 15.3, "Handling an Exception"

12.4 MSC53-CPP. Do not return from a function declared [[noreturn]]

The `[[noreturn]]` attribute specifies that a function does not return. The C++ Standard, [dcl.attr.noreturn] paragraph 2 [ISO/IEC 14882-2014], states the following:

> If a function `f` is called where `f` was previously declared with the `noreturn` attribute and `f` eventually returns, the behavior is undefined.

A function that specifies `[[noreturn]]` can prohibit returning by throwing an exception, entering an infinite loop, or calling another function designated with the `[[noreturn]]` attribute.

12.4.1 Noncompliant Code Example

In this noncompliant code example, if the value `0` is passed, control will flow off the end of the function, resulting in an implicit return and undefined behavior.

```
#include <cstdlib>

[[noreturn]] void f(int i) {
  if (i > 0)
    throw "Received positive input";
  else if (i < 0)
    std::exit(0);
}
```

12.4.2 Compliant Solution

In this compliant solution, the function does not return on any code path.

```
#include <cstdlib>

[[noreturn]] void f(int i) {
  if (i > 0)
    throw "Received positive input";
  std::exit(0);
}
```

12.4.3 Risk Assessment

Returning from a function marked `[[noreturn]]` results in undefined behavior that might be exploited to cause data-integrity violations.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
MSC53-CPP	Medium	Unlikely	Low	**P2**	**L3**

12.4.4 Bibliography

[ISO/IEC 14882-2014]	Subclause 7.6.3, “`noreturn` Attribute”

12.5 MSC54-CPP. A signal handler must be a plain old function

The C++ Standard, [support.runtime], paragraph 10 [ISO/IEC 14882-2014], states the following:

> The common subset of the C and C++ languages consists of all declarations, definitions, and expressions that may appear in a well formed C++ program and also in a conforming C program. A POF ("plain old function") is a function that uses only features from this common subset, and that does not directly or indirectly use any function that is not a POF, except that it may use plain lock-free atomic operations. A *plain lock-free atomic operation* is an invocation of a function *f* from Clause 29, such that *f* is not a member function, and either *f* is the function `atomic_is_lock_free`, or for every atomic argument A passed to *f*, `atomic_is_lock_free(A)` yields `true`. All signal handlers shall have C linkage. The behavior of any function other than a POF used as a signal handler in a C++ program is implementation-defined.228

Footnote 228 states the following:

> In particular, a signal handler using exception handling is very likely to have problems. Also, invoking `std::exit` may cause destruction of objects, including those of the standard library implementation, which, in general, yields undefined behavior in a signal handler.

If your signal handler is not a plain old function, then the behavior of a call to it in response to a signal is implementation-defined, at best, and is likely to result in undefined behavior. All signal handlers must meet the definition of a plain old function. In addition to the restrictions placed on signal handlers in a C program, this definition also prohibits the use of features that exist in C++ but not in C (such as non-POD [non–plain old data] objects and exceptions). This includes indirect use of such features through function calls.

12.5.1 Noncompliant Code Example

In this noncompliant code example, the signal handler is declared as a `static` function. However, since all signal handler functions must have C language linkage, and C++ is the default language linkage for functions in C++, calling the signal handler results in undefined behavior.

```
#include <csignal>

static void sig_handler(int sig) {
  // Implementation details elided.
}

void install_signal_handler() {
  if (SIG_ERR == std::signal(SIGTERM, sig_handler)) {
    // Handle error
  }
}
```

12.5.2 Compliant Solution

This compliant solution defines `sig_handler()` as having C language linkage. As a consequence of declaring the signal handler with C language linkage, the signal handler will have external linkage rather than internal linkage.

```
#include <csignal>

extern "C" void sig_handler(int sig) {
  // Implementation details elided.
}

void install_signal_handler() {
  if (SIG_ERR == std::signal(SIGTERM, sig_handler)) {
    // Handle error
  }
}
```

12.5.3 Noncompliant Code Example

In this noncompliant code example, a signal handler calls a function that allows exceptions, and it attempts to handle any exceptions thrown. Because exceptions are not part of the common subset of C and C++ features, this example results in implementation-defined behavior. However, it is unlikely that the implementation's behavior will be suitable. For instance, on a stack-based architecture where a signal is generated asynchronously (instead of as a result of a call to `std:abort()` or `std::raise()`), it is possible that the stack frame is not properly initialized, causing stack tracing to be unreliable and preventing the exception from being caught properly.

```
#include <csignal>

static void g() noexcept(false);

extern "C" void sig_handler(int sig) {
  try {
    g();
  } catch (...) {
    // Handle error
  }
}

void install_signal_handler() {
  if (SIG_ERR == std::signal(SIGTERM, sig_handler)) {
    // Handle error
  }
}
```

12.5.4 Compliant Solution

There is no compliant solution whereby `g()` can be called from the signal handler because it allows exceptions. Even if `g()` were implemented such that it handled all exceptions and was marked `noexcept(true)`, it would still be noncompliant to call `g()`from a signal handler because `g()` would still use a feature that is not a part of the common subset of C and C++ features allowed by a signal handler. Therefore, this compliant solution removes the call to `g()` from the signal handler and instead polls a variable of type `volatile sig_atomic_t` periodically; if the variable is set to `1` in the signal handler, then `g()` is called to respond to the signal.

```
#include <csignal>

volatile sig_atomic_t signal_flag = 0;
static void g() noexcept(false);

extern "C" void sig_handler(int sig) {
  signal_flag = 1;
}

void install_signal_handler() {
  if (SIG_ERR == std::signal(SIGTERM, sig_handler)) {
    // Handle error
  }
}

// Called periodically to poll the signal flag.
void poll_signal_flag() {
  if (signal_flag == 1) {
    signal_flag = 0;
    try {
      g();
    } catch(...) {
      // Handle error
    }
  }
}
```

12.5.5 Risk Assessment

Failing to use a plain old function as a signal handler can result in implementation-defined behavior as well as undefined behavior. Given the number of features that exist in C++ that do not also exist in C, the consequences that arise from failure to comply with this rule can range from benign (harmless) behavior to abnormal program termination, or even arbitrary code execution.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
MSC54-CPP	High	Probable	High	**P6**	**L2**

12.5.6 Related Guidelines

SEI CERT C Coding Standard	SIG30-C. Call only asynchronous-safe functions within signal handlers SIG31-C. Do not access shared objects in signal handlers

12.5.7 Bibliography

[ISO/IEC 14882-2014]	Subclause 18.10, “Other Runtime Support”