# Introduction

This standard provides rules for secure coding in the C++. The goal is to develop safe and reliable systems, for example by eliminating undefined behaviors that can lead to undefined program behaviors and exploitable vulnerabilities.

# Glossary

|  |  |
| --- | --- |
| **Term** | **Meaning** |
| Variadic function | Functions that accept more formal arguments at the call site than are specified by the parameter declaration clause. |
| cv-qualification | Refers to the use of "const" and "volatile" qualifiers applied to a type.   * const indicates that a variable's value cannot be modified after it is initialized. * volatile indicates that a variable's value may be changed by external factors such as hardware or other threads, and thus the compiler should not optimize access to the variable. |
| Standard-layout types | Type that can be used to communicate with code written in **other programming languages**, as the layout of the type is strictly specified.  For example, a *standard-layout class* is 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,. * Recursively, does not have nonstatic data members of nonstandard-layout type. |
| Nonstandard-layout types |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

# Most-Common Rule Summary

NOTE: This is my self-defined rule summary for quick memory and code review. It’s not available in MISRA official document.

|  |  |
| --- | --- |
| **Rule** | **Simple Explanation** |
|  |  |
|  |  |
|  |  |
|  |  |

Declarations and Initialization (DCL)

Rule DCL50-CPP: Do not define a C-style variadic function

Rule DCL51-CPP: Do not declare or define a reserved identifier

Rule DCL52-CPP: Never qualify a reference type with const or volatile

Rule DCL53-CPP: Do not write syntactically ambiguous declarations

Rule DCL54-CPP: Overload allocation and deallocation functions as a pair in the same scope

Rule DCL55-CPP: Avoid information leakage when passing a class object across a trust boundary

Rule DCL56-CPP: Avoid cycles during initialization of static objects

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

Rule DCL58-CPP: Do not modify the standard namespaces

Rule DCL59-CPP: Do not define an unnamed namespace in a header file

Rule DCL60-CPP: Obey the one-definition rule

Expressions (EXP)

Rule EXP50-CPP: Do not depend on the order of evaluation for side effects

Rule EXP51-CPP: Do not delete an array through a pointer of the incorrect type

Rule EXP52-CPP: Do not rely on side effects in unevaluated operands

Rule EXP53-CPP: Do not read uninitialized memory

Rule EXP54-CPP: Do not access an object outside of its lifetime

Rule EXP55-CPP: Do not access a cv-qualified object through a cv-unqualified type

Rule EXP56-CPP: Do not call a function with a mismatched language linkage

Rule EXP57-CPP: Do not cast or delete pointers to incomplete classes

Rule EXP58-CPP: Pass an object of the correct type to va\_start

Rule EXP59-CPP: Use offsetof() on valid types and members

Rule EXP60-CPP: Do not pass a nonstandard-layout type object across execution boundaries

Rule EXP61-CPP: A lambda object must not outlive any of its reference captured objects

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

Rule EXP63-CPP: Do not rely on the value of a moved-from object

Integers (INT)

Rule INT50-CPP: Do not cast to an out-of-range enumeration value

5 Containers (CTR)

Rule CTR50-CPP: Guarantee that container indices and iterators are within the valid range

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

Rule CTR52-CPP: Guarantee that library functions do not overflow

Rule CTR53-CPP: Use valid iterator ranges

Rule CTR54-CPP: Do not subtract iterators that do not refer to the same container

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

Rule CTR56-CPP: Do not use pointer arithmetic on polymorphic objects

Rule CTR57-CPP: Provide a valid ordering predicate

Rule CTR58-CPP: Predicate function objects should not be mutable

6 Characters and Strings (STR)

Rule STR50-CPP: Guarantee that storage for strings has sufficient space for character data and the null terminator

Rule STR51-CPP: Do not attempt to create a std::string from a null pointer

Rule STR52-CPP: Use valid references, pointers, and iterators to reference elements of a basic\_string

Rule STR53-CPP: Range check element access

7 Memory Management (MEM)

Rule MEM50-CPP: Do not access freed memory

Rule MEM51-CPP: Properly deallocate dynamically allocated resources

Rule MEM52-CPP: Detect and handle memory allocation errors

Rule MEM53-CPP: Explicitly construct and destruct objects when manually managing object lifetime

Rule MEM54-CPP: Provide placement new with properly aligned pointers to sufficient storage capacity

Rule MEM55-CPP: Honor replacement dynamic storage management requirements

Rule MEM56-CPP: Do not store an already-owned pointer value in an unrelated smart pointer

Rule MEM57-CPP: Avoid using default operator new for over-aligned types

8 Input Output (FIO)

Rule FIO50-CPP: Do not alternately input and output from a file stream without an intervening positioning call

Rule FIO51-CPP: Close files when they are no longer needed

9 Exceptions and Error Handling (ERR)

Rule ERR50-CPP: Do not abruptly terminate the program

Rule ERR51-CPP: Handle all exceptions

Rule ERR52-CPP: Do not use setjmp() or longjmp()

Rule ERR53-CPP: Do not reference base classes or class data members in a constructor or destructor function-try-block handler

Rule ERR54-CPP: Catch handlers should order their parameter types from most derived to least derived

Rule ERR55-CPP: Honor exception specifications

Rule ERR56-CPP: Guarantee exception safety

Rule ERR57-CPP: Do not leak resources when handling exceptions

Rule ERR58-CPP: Handle all exceptions thrown before main() begins executing

Rule ERR59-CPP: Do not throw an exception across execution boundaries

Rule ERR60-CPP: Exception objects must be nothrow copy constructible

Rule ERR61-CPP: Catch exceptions by lvalue reference

Rule ERR62-CPP: Detect errors when converting a string to a number

10 Object Oriented Programming (OOP)

Rule OOP50-CPP: Do not invoke virtual functions from constructors or destructors

Rule OOP51-CPP: Do not slice derived objects

Rule OOP52-CPP: Do not delete a polymorphic object without a virtual destructor

Rule OOP53-CPP: Write constructor member initializers in the canonical order

Rule OOP54-CPP: Gracefully handle self-copy assignment

Rule OOP55-CPP: Do not use pointer-to-member operators to access nonexistent members

Rule OOP56-CPP: Honor replacement handler requirements

Rule OOP57-CPP: Prefer special member functions and overloaded operators to C Standard Library functions

Rule OOP58-CPP: Copy operations must not mutate the source object

11 Concurrency (CON)

Rule CON50-CPP: Do not destroy a mutex while it is locked

Rule CON51-CPP: Ensure actively held locks are released on exceptional conditions

Rule CON52-CPP: Prevent data races when accessing bit-fields from multiple threads

Rule CON53-CPP: Avoid deadlock by locking in a predefined order

Rule CON54-CPP: Wrap functions that can spuriously wake up in a loop

Rule CON55-CPP: Preserve thread safety and liveness when using condition variables

Rule CON56-CPP: Do not speculatively lock a non-recursive mutex that is already owned by the calling thread

12 Miscellaneous (MSC)

Rule MSC50-CPP: Do not use std::rand() for generating pseudorandom numbers

Rule MSC51-CPP: Ensure your random number generator is properly seeded

Rule MSC52-CPP: Value-returning functions must return a value from all exit paths

Rule MSC53-CPP: Do not return from a function declared [[noreturn]]

Rule MSC54-CPP: A signal handler must be a plain old function

# Rules

## Declarations and Initialization (DCL)

### Rule DCL50-CPP: Do not define a C-style variadic function

Reason: C-style variadic functions have 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. This could lead to undefined behavior.

|  |  |
| --- | --- |
| **Non-compliant** | **Compliant** |
| #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;  } | #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 solution does not result in undefined behavior if the list of parameters is not terminated with 0. Also, if any of the values passed to the function are not integers, the code is ill-formed rather than producing undefined behavior |

**MISRA identical rule**: Rule 8-4-1 – Functions shall not be defined using the ellipsis notation.

### Rule DCL51-CPP: Do not declare or define a *reserved identifier*

<https://wiki.sei.cmu.edu/confluence/display/cplusplus/DCL51-CPP.+Do+not+declare+or+define+a+reserved+identifier>

|  |  |
| --- | --- |
| **Non-compliant** | **Compliant** |
| #ifndef \_MY\_HEADER\_H\_  #define \_MY\_HEADER\_H\_  // Contents of <my\_header.h>  #endif // \_MY\_HEADER\_H\_ | #ifndef MY\_HEADER\_H  #define MY\_HEADER\_H  // Contents of <my\_header.h>  #endif // MY\_HEADER\_H  This compliant solution avoids using leading or trailing underscores in the name of the header guard. |
| #include <cinttypes> // for int\_fast16\_t  void f(std::int\_fast16\_t val) {  enum { MAX\_SIZE = 80 };  // ...  } | #include <cinttypes> // for std::int\_fast16\_t  void f(std::int\_fast16\_t val) {  enum { BufferSize = 80 };  // ...  }  Header <cinttypes> is specified to include <cstdint>. MAX\_SIZE conflicts with the name of the <cstdint> header macro used to denote the upper limit of std:size\_t. |

**MISRA idential rule**: Rule 17-0-1 – Reserved identifiers, macros and functions in the standard library shall not be defined, redefined or undefined.

### Rule DCL52-CPP: Never qualify a reference type with *const* or *volatile*

Reason: 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.

|  |  |
| --- | --- |
| **Non-compliant** | **Compliant** |
| char c = 'a';  char &const p = c; | char c = 'a';  char &p = c;  char c = 'a';  char const &p = c;  char c = 'a';  const char &p = c; |

### Rule DCL53-CPP: Do not write syntactically ambiguous declarations

|  |  |
| --- | --- |
| **Non-compliant** | **Compliant** |
| #include <iostream>  struct Widget {  Widget() { std::cout << "Constructed" << std::endl; }  };  void f() {  Widget w();  }  Declaration of w is syntactically ambiguous. It could be either a declaration of a function pointer with no arguments and returning a Widget, or a declaration of a local variable of type Widget. The syntax used here defines the former instead of the latter.  As a result, this program compiles and prints no output because the default constructor is never invoked. | #include <iostream>  struct Widget {  Widget() { std::cout << "Constructed" << std::endl; }  };  void f() {  Widget w1; // Use no parentheses  Widget w2{}; // Use direct initialization  }  This code produces the output Constructed twice, once for w1 and once for w2. |

### Rule DCL54-CPP: Overload allocation and deallocation functions as a pair in the same scope

|  |  |
| --- | --- |
| **Non-compliant** | **Compliant** |
|  |  |

### Rule DCL55-CPP: Avoid information leakage when passing a class object across a trust boundary

|  |  |
| --- | --- |
| **Non-compliant** | **Compliant** |
|  |  |

### Rule DCL56-CPP: Avoid cycles during initialization of *static* objects

### Rule DCL57-CPP: Do not let exceptions escape from destructors or deallocation functions

### Rule DCL58-CPP: Do not modify the standard namespaces

|  |  |
| --- | --- |
| **Non-compliant** | **Compliant** |
| namespace std {  int x;  } | namespace nonstd {  int x;  } |

### Rule DCL59-CPP: Do not define an unnamed namespace in a header file

**MISRA identical rule**: Rule 7-3-3 – There shall be no unnamed namespaces in header files

|  |  |
| --- | --- |
| **Non-compliant** | **Compliant** |
| // Header.hpp  namespace { // Non-compliant  extern int32\_t x;  } | // Header.hpp  namespace ABC { // Compliant  extern int32\_t x;  } |

### Rule DCL60-CPP: Obey the one-definition rule (ODR)

|  |  |
| --- | --- |
| **Non-compliant** | **Compliant** |
| // a.cpp  struct S {    int a;  };    // b.cpp  class S {  public:    int a;  }; | // S.h  struct S {    int a;  };    // a.cpp  #include "S.h"    // b.cpp  #include "S.h" |

**MISRA identical rule**: Rule 3-2-2 – The One Definition Rule shall not be violated.

## Expressions (EXP)

### Rule EXP50-CPP: Do not depend on the order of evaluation for side effects

|  |  |
| --- | --- |
| **Non-compliant** | **Compliant** |
| void f(int i, const int \*b) {    int a = i + b[++i];    // ...  } | void f(int i, const int \*b) {    ++i;    int a = i + b[i];    // ...  } |
| #include <iostream>    void f(int i) {  std::cout << i++ << i << std::endl;  } | #include <iostream>    void f(int i) {  std::cout << i++;  std::cout << i << std::endl;  } |

### Rule EXP51-CPP: Do not delete an array through a pointer of the incorrect type

|  |  |
| --- | --- |
| **Non-compliant** | **Compliant** |
| struct Base {    virtual ~Base() = default;  };    struct Derived final : Base {};    void f() {     Base \*b = new Derived[10];     // ...     delete [] b;  } | struct Base {    virtual ~Base() = default;  };    struct Derived final : Base {};    void f() {     Derived \*b = new Derived[10];     // ...     delete [] b;  } |

### Rule EXP52-CPP: Do not rely on side effects in unevaluated operands

### Rule EXP53-CPP: Do not read uninitialized memory

|  |  |
| --- | --- |
| **Non-compliant** | **Compliant** |
| #include <iostream>    void f() {    int i;    std::cout << i;  } | #include <iostream>    void f() {    int i = 0;    std::cout << i;  } |
| #include <iostream>    void f() {    int \*i = new int;    std::cout << i << ", " << \*i;  } | #include <iostream>    void f() {    int \*i = new int(12);    std::cout << i << ", " << \*i;  } |
| class S {    int c;    public:    int f(int i) const { return i + c; }  };    void f() {    S s;    int i = s.f(10);  } | 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);  } |

### Rule EXP54-CPP: Do not access an object outside of its lifetime

### Rule EXP55-CPP: Do not access a cv-qualified object through a cv-unqualified type

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

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.

|  |  |
| --- | --- |
| **Non-compliant** | **Compliant** |
| void g(const int &ci) {    int &ir = const\_cast<int &>(ci);    ir = 42;  }    void f() {    const int i = 4;    g(i);  } | void g(int &i) {    i = 42;  }    void f() {    int i = 4;    g(i);  } |

**MISRA identical rule**: Rule 5-2-5 – A cast shall not remove any *const* or *volatile* type of a pointer or reference

### Rule EXP56-CPP: Do not call a function with a mismatched language linkage

### Rule EXP57-CPP: Do not cast or delete pointers to incomplete classes

### Rule EXP58-CPP: Pass an object of the correct type to va\_start

### Rule EXP59-CPP: Use offsetof() on valid types and members

### Rule EXP60-CPP: Do not pass a nonstandard-layout type object across execution boundaries

### Rule EXP61-CPP: A lambda object must not outlive any of its reference captured objects

### Rule EXP62-CPP: Do not access the bits of an object representation that are not part of the object’s value representation

### Rule EXP63-CPP: Do not rely on the value of a moved-from object

## Integers (INT)

### Rule INT50-CPP: Do not cast to an out-of-range enumeration value

## Containers (CTR)

### Rule CTR50-CPP: Guarantee that container indices and iterators are within the valid range

### Rule CTR51-CPP: Use valid references, pointers, and iterators to reference elements of a container

### Rule CTR52-CPP: Guarantee that library functions do not overflow

### Rule CTR53-CPP: Use valid iterator ranges

### Rule CTR54-CPP: Do not subtract iterators that do not refer to the same container

### Rule CTR55-CPP: Do not use an additive operator on an iterator if the result would overflow

### Rule CTR56-CPP: Do not use pointer arithmetic on polymorphic objects

### Rule CTR57-CPP: Provide a valid ordering predicate

### Rule CTR58-CPP: Predicate function objects should not be mutable

## Characters and Strings (STR)

### Rule STR50-CPP: Guarantee that storage for strings has sufficient space for character data and the null terminator

### Rule STR51-CPP: Do not attempt to create a std::string from a null pointer

### Rule STR52-CPP: Use valid references, pointers, and iterators to reference elements of a basic\_string

### Rule STR53-CPP: Range check element access

## Memory Management (MEM)

### Rule MEM50-CPP: Do not access freed memory

### Rule MEM51-CPP: Properly deallocate dynamically allocated resources

### Rule MEM52-CPP: Detect and handle memory allocation errors

### Rule MEM53-CPP: Explicitly construct and destruct objects when manually managing object lifetime

### Rule MEM54-CPP: Provide placement new with properly aligned pointers to sufficient storage capacity

### Rule MEM55-CPP: Honor replacement dynamic storage management requirements

### Rule MEM56-CPP: Do not store an already-owned pointer value in an unrelated smart pointer

### Rule MEM57-CPP: Avoid using default operator new for over-aligned types

## Input Output (FIO)

### Rule FIO50-CPP: Do not alternately input and output from a file stream without an intervening positioning call

### Rule FIO51-CPP: Close files when they are no longer needed

## Exceptions and Error Handling (ERR)

### Rule ERR50-CPP: Do not abruptly terminate the program

### Rule ERR51-CPP: Handle all exceptions

### Rule ERR52-CPP: Do not use setjmp() or longjmp()

### Rule ERR53-CPP: Do not reference base classes or class data members in a constructor or destructor function-try-block handler

### Rule ERR54-CPP: Catch handlers should order their parameter types from most derived to least derived

### Rule ERR55-CPP: Honor exception specifications

### Rule ERR56-CPP: Guarantee exception safety

### Rule ERR57-CPP: Do not leak resources when handling exceptions

### Rule ERR58-CPP: Handle all exceptions thrown before main() begins executing

### Rule ERR59-CPP: Do not throw an exception across execution boundaries

### Rule ERR60-CPP: Exception objects must be nothrow copy constructible

### Rule ERR61-CPP: Catch exceptions by lvalue reference

### Rule ERR62-CPP: Detect errors when converting a string to a number

## Object Oriented Programming (OOP)

### Rule OOP50-CPP: Do not invoke virtual functions from constructors or destructors

### Rule OOP51-CPP: Do not slice derived objects

### Rule OOP52-CPP: Do not delete a polymorphic object without a virtual destructor

### Rule OOP53-CPP: Write constructor member initializers in the canonical order

### Rule OOP54-CPP: Gracefully handle self-copy assignment

### Rule OOP55-CPP: Do not use pointer-to-member operators to access nonexistent members

### Rule OOP56-CPP: Honor replacement handler requirements

### Rule OOP57-CPP: Prefer special member functions and overloaded operators to C Standard Library functions

### Rule OOP58-CPP: Copy operations must not mutate the source object

## Concurrency (CON)

### Rule CON50-CPP: Do not destroy a mutex while it is locked

### Rule CON51-CPP: Ensure actively held locks are released on exceptional conditions

### Rule CON52-CPP: Prevent data races when accessing bit-fields from multiple threads

### Rule CON53-CPP: Avoid deadlock by locking in a predefined order

### Rule CON54-CPP: Wrap functions that can spuriously wake up in a loop

### Rule CON55-CPP: Preserve thread safety and liveness when using condition variables

### Rule CON56-CPP: Do not speculatively lock a non-recursive mutex that is already owned by the calling thread

## Miscellaneous (MSC)

### Rule MSC50-CPP: Do not use std::rand() for generating pseudorandom numbers

### Rule MSC51-CPP: Ensure your random number generator is properly seeded

### Rule MSC52-CPP: Value-returning functions must return a value from all exit paths

### Rule MSC53-CPP: Do not return from a function declared [[noreturn]]

### Rule MSC54-CPP: A signal handler must be a plain old function