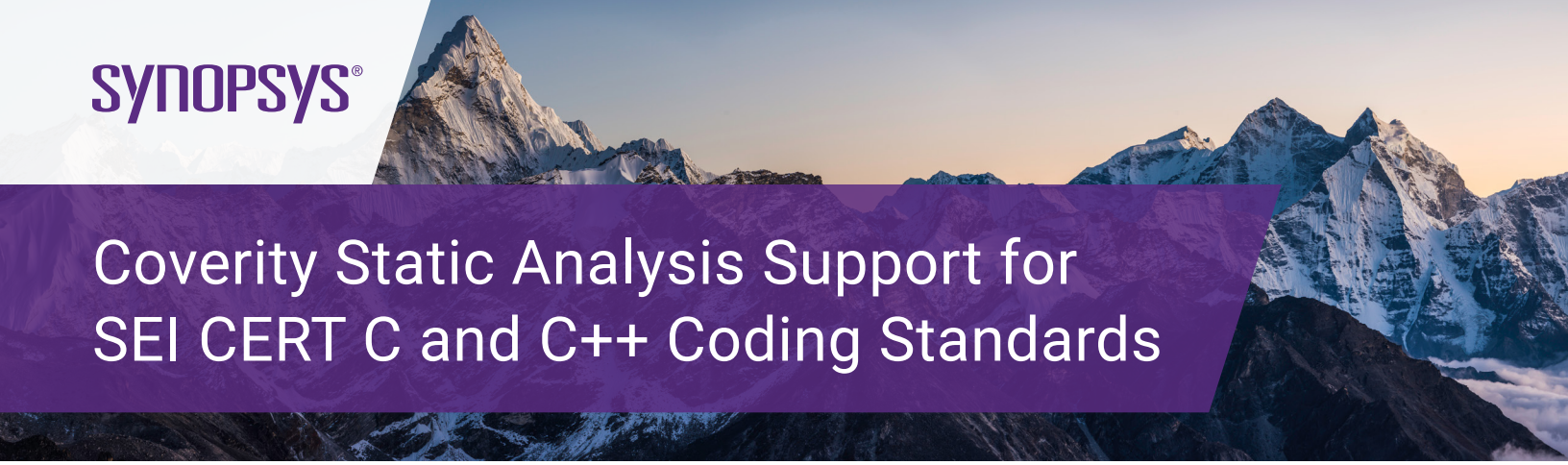
SYNOPSYS®



Coverity Static Analysis Support for SEI CERT C and C++ Coding Standards

Fully ensure the safety, reliability, and security of software written in C and C++

The SEI CERT C and C++ Coding Standards are lists of rules for writing secure code in the C and C++ programming languages. They represent an important milestone in introducing best practices for ensuring the safety, reliability, security, and integrity of software written in C/C++. Notably, the standards are designed to be enforceable by software code analyzers using static analysis techniques. This greatly reduces the cost of compliance by way of automation.

Adhering to coding standards is a crucial step in establishing best coding practices. Standards adherence is particularly important in safety-critical, high-impact industries, such as automotive, medical, and networking. Software defects in products coming from these industries manifest themselves physically and tangibly—often with life-threatening consequences.

Synopsys provides a comprehensive solution for the SEI CERT C/C++ Coding Standards. Coverity Static Analysis implements the Rules category within the CERT C/C++ standards, as well as methods for managing violations and reporting on them.

SEI CERT C Coding Standard (2016 Edition)

The SEI CERT C Coding Standard was developed specifically for the following versions of the C language:

- ISO/IEC 9899:2011: Information Technology—Programming Languages—C, 3rd ed.
- ISO/IEC 9899:2011/Cor 1:2012: Information Technology—Programming Languages—C Technical Corrigendum 1

These versions are commonly referred to as the C11 standard. The CERT C rules may also be applied to earlier versions of the C language, such as C99.

The 2016 edition of the CERT C standard contains 99 coding rules and reflects the C rules available on the CERT Secure Coding wiki as of March 30, 2016. The CERT Secure Coding wiki for C is here:

https://www.securecoding.cert.org/confluence/display/c/

The SEI CERT C Coding Standard (2016 Edition) is here:

https://www.cert.org/secure-coding/products-services/secure-coding-download.cfm

The CERT C wiki also documents 185 recommendations and two platform-specific annexes (POSIX and Windows). The recommendations and annexes are not part of the core secure coding standard.

The “Level” column in the “supported rules” tables below indicates the severity, remediation cost, and likelihood of each vulnerability. Level 1 vulnerabilities are the most severe, the least expensive to repair, and/or the most likely to be found in code. Meanwhile, Level 3 vulnerabilities are the least severe, the most expensive to repair, and/or the least likely to be found in code.

CERT C rule coverage

Section	Rules		% Coverage
	Supported	All	
All	**99**	**99**	**100.0**
PRE	3	3	100.0
DCL	8	8	100.0
EXP	14	14	100.0
INT	7	7	100.0
FLP	5	5	100.0
ARR	6	6	100.0
STR	6	6	100.0
MEM	6	6	100.0
FIO	13	13	100.0
ENV	5	5	100.0
SIG	4	4	100.0
ERR	3	3	100.0
CON	12	12	100.0
MSC	7	7	100.0

CERT C supported rules

Rule	Description	Level
PRE30-C	Do not create a universal character name through concatenation.	L3
PRE31-C	Avoid side effects in arguments to unsafe macros.	L3
PRE32-C	Do not use preprocessor directives in invocations of function-like macros.	L3
DCL30-C	Declare objects with appropriate storage durations.	L2
DCL31-C	Declare identifiers before using them.	L3
DCL36-C	Do not declare an identifier with conflicting linkage classifications.	L2
DCL37-C	Do not declare or define a reserved identifier.	L3
DCL38-C	Use the correct syntax when declaring a flexible array member.	L3
DCL39-C	Avoid information leakage when passing a structure across a trust boundary.	L3
DCL40-C	Do not create incompatible declarations of the same function or object.	L3
DCL41-C	Do not declare variables inside a switch statement before the first case label.	L3
EXP30-C	Do not depend on order of evaluation for side effects.	L2
EXP32-C	Do not access a volatile object through a nonvolatile reference.	L2

Rule	Description	Level
EXP33-C	Do not read uninitialized memory.	L1
EXP34-C	Do not dereference null pointers.	L1
EXP35-C	Do not modify objects with temporary lifetime.	L3
EXP36-C	Do not cast pointers into more strictly aligned pointer types.	L3
EXP37-C	Call functions with the correct number and type of arguments.	L3
EXP39-C	Do not access a variable through a pointer of an incompatible type.	L3
EXP40-C	Do not modify constant objects.	L3
EXP42-C	Do not compare padding data.	L2
EXP43-C	Avoid undefined behavior when using restrict-qualified pointers.	L3
EXP44-C	Do not rely on side effects in operands to sizeof, _Alignof, or _Generic.	L3
EXP45-C	Do not perform assignments in selection statements.	L2
EXP46-C	Do not use a bitwise operator with a Boolean-like operand.	L2
INT30-C	Ensure that unsigned integer operations do not wrap.	L2
INT31-C	Ensure that integer conversions do not result in lost or misinterpreted data.	L2
INT32-C	Ensure that operations on signed integers do not result in overflow.	L2
INT33-C	Ensure that division and modulo operations do not result in divide-by-zero errors.	L2
INT34-C	Do not shift an expression by a negative number of bits or by greater than or equal to the number of bits that exist in the operand.	L3
INT35-C	Use correct integer precisions.	L3
INT36-C	Converting a pointer to integer or integer to pointer.	L3
FLP30-C	Do not use floating-point variables as loop counters.	L2
FLP32-C	Prevent or detect domain and range errors in math functions.	L2
FLP34-C	Ensure that floating-point conversions are within range of the new type.	L3
FLP36-C	Preserve precision when converting integral values to floating-point type.	L3
FLP37-C	Do not use object representations to compare floating-point values.	L3
ARR30-C	Do not form or use out of bounds pointers or array subscripts.	L2
ARR32-C	Ensure size arguments for variable length arrays are in a valid range.	L2
ARR36-C	Do not subtract or compare two pointers that do not refer to the same array.	L2
ARR37-C	Do not add or subtract an integer to a pointer to a non-array object.	L2
ARR38-C	Guarantee that library functions do not form invalid pointers.	L1
ARR39-C	Do not add or subtract a scaled integer to a pointer.	L2
STR30-C	Do not attempt to modify string literals.	L2
STR31-C	Guarantee that storage for strings has sufficient space for character data and the null terminator.	L1
STR32-C	Do not pass a non-null-terminated character sequence to a library function that expects a string.	L1

Rule	Description	Level
STR34-C	Cast characters to unsigned char before converting to larger integer sizes.	L2
STR37-C	Arguments to character-handling functions must be representable as an unsigned char.	L3
STR38-C	Do not confuse narrow and wide character strings and functions.	L1
MEM30-C	Do not access freed memory.	L1
MEM31-C	Free dynamically allocated memory when no longer needed.	L2
MEM33-C	Allocate and copy structures containing a flexible array member dynamically.	L3
MEM34-C	Only free memory allocated dynamically.	L1
MEM35-C	Allocate sufficient memory for an object.	L2
MEM36-C	Do not modify the alignment of objects by calling realloc().	L3
FIO30-C	Exclude user input from format strings.	L1
FIO32-C	Do not perform operations on devices that are only appropriate for files.	L3
FIO34-C	Distinguish between characters read from a file and EOF or WEOF.	L1
FIO37-C	Do not assume that fgets() or fgetws() returns a nonempty string when successful.	L1
FIO38-C	Do not copy a FILE object.	L3
FIO39-C	Do not alternately input and output from a stream without an intervening flush or positioning call.	L2
FIO40-C	Reset strings on fgets() or fgetws() failure.	L3
FIO41-C	Do not call getc(), putc(), getwc(), or putwc() with a stream argument that has side effects.	L3
FIO42-C	Close files when they are no longer needed.	L3
FIO44-C	Only use values for fsetpos() that are returned from fgetpos().	L3
FIO45-C	Avoid TOCTOU race conditions while accessing files.	L2
FIO46-C	Do not access a closed file.	L3
FIO47-C	Use valid format strings.	L2
ENV30-C	Do not modify the object referenced by the return value of certain functions.	L3
ENV31-C	Do not rely on an environment pointer following an operation that may invalidate it.	L3
ENV32-C	All exit handlers must return normally.	L1
ENV33-C	Do not call system().	L1
ENV34-C	Do not store pointers returned by certain functions.	L3
SIG30-C	Call only asynchronous-safe functions within signal handlers.	L1
SIG31-C	Do not access shared objects in signal handlers.	L2
SIG34-C	Do not call signal() from within interruptible signal handlers.	L3
SIG35-C	Do not return from a computational exception signal handler.	L3
ERR30-C	Set errno to zero before calling a library function known to set errno, and check errno only after the function returns a value indicating failure.	L2
ERR32-C	Do not rely on indeterminate values of errno.	L3

Rule	Description	Level
ERR33-C	Detect and handle standard library errors.	L1
CON30-C	Clean up thread-specific storage.	L3
CON31-C	Do not destroy a mutex while it is locked.	L3
CON32-C	Prevent data races when accessing bit-fields from multiple threads.	L2
CON33-C	Avoid race conditions when using library functions.	L3
CON34-C	Declare objects shared between threads with appropriate storage durations.	L3
CON35-C	Avoid deadlock by locking in predefined order.	L3
CON36-C	Wrap functions that can spuriously wake up in a loop.	L3
CON37-C	Do not call signal() in a multithreaded program.	L2
CON38-C	Preserve thread safety and liveness when using condition variables.	L3
CON39-C	Do not join or detach a thread that was previously joined or detached.	L2
CON40-C	Do not refer to an atomic variable twice in an expression.	L2
CON41-C	Wrap functions that can fail spuriously in a loop.	L3
MSC30-C	Do not use the rand() function for generating pseudorandom numbers.	L2
MSC32-C	Properly seed pseudorandom number generators.	L1
MSC33-C	Do not pass invalid data to the asctime() function.	L1
MSC37-C	Ensure that control never reaches the end of a non-void function.	L2
MSC38-C	Do not treat a predefined identifier as an object if it might only be implemented as a macro.	L3
MSC39-C	Do not call va_arg() on a va_list that has an indeterminate value.	L3
MSC40-C	Do not violate constraints.	L3

SEI CERT C++ Coding Standard (2016 Edition)

The SEI CERT C++ Coding Standard was developed specifically for the following version of the C++ language:

- ISO/IEC 14882-2014: Information Technology---Programming Languages—C++, 4th ed.

This version is commonly referred to as the C++14 standard. The CERT C++ rules may also be applied to earlier versions of the C++ language, such as C++11.

The 2016 edition of the CERT C++ standard contains 83 coding rules. The CERT Secure Coding wiki for C++ is here:

https://www.securecoding.cert.org/confluence/display/cplusplus/

The SEI CERT C++ Coding Standard (2016 Edition) is here:

http://cert.org/secure-coding/products-services/secure-coding-cpp-download-2016.cfm

The CERT C++ wiki also documents some recommendations. The recommendations are not part of the core secure coding standard.

CERT C++ rule coverage

Section	Rules		% Coverage
	Supported	All	
All	**83**	**83**	**100.0**
DCL	11	11	100.0
EXP	14	14	100.0
INT	1	1	100.0
CTR	9	9	100.0
STR	4	4	100.0
MEM	8	8	100.0
FIO	2	2	100.0
ERR	13	13	100.0
OOP	9	9	100.0
CON	7	7	100.0
MSC	5	5	100.0

CERT C++ supported rules

Rule	Description	Level
DCL50-CPP	Do not define a C-style variadic function.	L1
DCL51-CPP	Do not declare or define a reserved identifier.	L3
DCL52-CPP	Never qualify a reference type with const or volatile.	L3
DCL53-CPP	Do not write syntactically ambiguous declarations.	L3
DCL54-CPP	Overload allocation and deallocation functions as a pair in the same scope.	L2
DCL55-CPP	Avoid information leakage when passing a class object across a trust boundary.	L3
DCL56-CPP	Avoid cycles during initialization of static objects.	L3
DCL57-CPP	Do not let exceptions escape from destructors or deallocation functions.	L3
DCL58-CPP	Do not modify the standard namespaces.	L2
DCL59-CPP	Do not define an unnamed namespace in a header file.	L3
DCL60-CPP	Obey the one-definition rule.	L3
EXP50-CPP	Do not depend on the order of evaluation for side effects.	L2
EXP51-CPP	Do not delete an array through a pointer of the incorrect type.	L3
EXP52-CPP	Do not rely on side effects in unevaluated operands.	L3
EXP53-CPP	Do not read uninitialized memory.	L1
EXP54-CPP	Do not access an object outside of its lifetime.	L2
EXP55-CPP	Do not access a cv-qualified object through a cv-unqualified type.	L2
EXP56-CPP	Do not call a function with a mismatched language linkage.	L3
EXP57-CPP	Do not cast or delete pointers to incomplete classes.	L3
EXP58-CPP	Pass an object of the correct type to va_start.	L3
EXP59-CPP	Use offsetof() on valid types and members.	L3

Rule	Description	Level
EXP60-CPP	Do not pass a nonstandard-layout type object across execution boundaries.	L1
EXP61-CPP	A lambda object must not outlive any of its reference captured objects.	L2
EXP62-CPP	Do not access the bits of an object representation that are not part of the object's value representation.	L2
EXP63-CPP	Do not rely on the value of a moved-from object.	L2
INT50-CPP	Do not cast to an out-of-range enumeration value.	L3
CTR50-CPP	Guarantee that container indices and iterators are within the valid range.	L2
CTR51-CPP	Use valid references, pointers, and iterators to reference elements of a container.	L2
CTR52-CPP	Guarantee that library functions do not overflow.	L1
CTR53-CPP	Use valid iterator ranges.	L2
CTR54-CPP	Do not subtract iterators that do not refer to the same container.	L2
CTR55-CPP	Do not use an additive operator on an iterator if the result would overflow.	L1
CTR56-CPP	Do not use pointer arithmetic on polymorphic objects.	L2
CTR57-CPP	Provide a valid ordering predicate.	L3
CTR58-CPP	Predicate function objects should not be mutable.	L3
STR50-CPP	Guarantee that storage for strings has sufficient space for character data and the null terminator.	L1
STR51-CPP	Do not attempt to create a std::string from a null pointer.	L1
STR52-CPP	Use valid references, pointers, and iterators to reference elements of a basic_string.	L2
STR53-CPP	Range check element access.	L2
MEM50-CPP	Do not access freed memory.	L1
MEM51-CPP	Properly deallocate dynamically allocated resources.	L1
MEM52-CPP	Detect and handle memory allocation errors.	L1
MEM53-CPP	Explicitly construct and destruct objects when manually managing object lifetime.	L1
MEM54-CPP	Provide placement new with properly aligned pointers to sufficient storage capacity.	L1
MEM55-CPP	Honor replacement dynamic storage management requirements.	L1
MEM56-CPP	Do not store an already-owned pointer value in an unrelated smart pointer.	L1
MEM57-CPP	Avoid using default operator new for over-aligned types.	L2
FIO50-CPP	Do not alternately input and output from a file stream without an intervening positioning call.	L2
FIO51-CPP	Close files when they are no longer needed.	L3
ERR50-CPP	Do not abruptly terminate the program.	L3
ERR51-CPP	Handle all exceptions.	L3
ERR52-CPP	Do not use setjmp() or longjmp().	L3
ERR53-CPP	Do not reference base classes or class data members in a constructor or destructor function-try-block handler.	L3
ERR54-CPP	Catch handlers should order their parameter types from most derived to least derived.	L1
ERR55-CPP	Honor exception specifications.	L2
ERR56-CPP	Guarantee exception safety.	L2

Rule	Description	Level
ERR57-CPP	Do not leak resources when handling exceptions.	L3
ERR58-CPP	Handle all exceptions thrown before main() begins executing.	L2
ERR59-CPP	Do not throw an exception across execution boundaries.	L1
ERR60-CPP	Exception objects must be nothrow copy constructible.	L3
ERR61-CPP	Catch exceptions by lvalue reference.	L3
ERR62-CPP	Detect errors when converting a string to a number.	L3
OOP50-CPP	Do not invoke virtual functions from constructors or destructors.	L3
OOP51-CPP	Do not slice derived objects.	L3
OOP52-CPP	Do not delete a polymorphic object without a virtual destructor.	L2
OOP53-CPP	Write constructor member initializers in the canonical order.	L3
OOP54-CPP	Gracefully handle self-copy assignment.	L3
OOP55-CPP	Do not use pointer-to-member operators to access nonexistent members.	L2
OOP56-CPP	Honor replacement handler requirements.	L3
OOP57-CPP	Prefer special member functions and overloaded operators to C Standard Library functions.	L2
OOP58-CPP	Copy operations must not mutate the source object.	L2
CON50-CPP	Do not destroy a mutex while it is locked.	L3
CON51-CPP	Ensure actively held locks are released on exceptional conditions.	L2
CON52-CPP	Prevent data races when accessing bit-fields from multiple threads.	L2
CON53-CPP	Avoid deadlock by locking in a predefined order.	L3
CON54-CPP	Wrap functions that can spuriously wake up in a loop.	L3
CON55-CPP	Preserve thread safety and liveness when using condition variables.	L3
CON56-CPP	Do no speculatively lock a non-recursive mutex that is already owned by the calling thread.	L3
MSC50-CPP	Do not use std::rand() for generating pseudorandom numbers.	L2
MSC51-CPP	Ensure your random number generator is properly seeded.	L1
MSC52-CPP	Value-returning functions must return a value from all exit paths.	L2
MSC53-CPP	Do not return from a function declared [[noreturn]].	L3
MSC54-CPP	A signal handler must be a plain old function.	L2

The Synopsys difference

Synopsys helps development teams build secure, high-quality software, minimizing risks while maximizing speed and productivity. Synopsys, a recognized leader in application security, provides static analysis, software composition analysis, and dynamic analysis solutions that enable teams to quickly find and fix vulnerabilities and defects in proprietary code, open source components, and application behavior. With a combination of industry-leading tools, services, and expertise, only Synopsys helps organizations optimize security and quality in DevSecOps and throughout the software development life cycle.

For more information, go to www.synopsys.com/software.

Synopsys, Inc.
185 Berry Street, Suite 6500
San Francisco, CA 94107 USA

U.S. Sales: 800.873.8193
International Sales: +1 415.321.5237
Email: sig-info@synopsys.com

SYNOPSYS®
Silicon to Software™

05/30/18.DS_StaticAnalysisCERTC_052418.