ISO/IEC JTC 1/SC 22/WG23 N1012

Date: 2020-11-23

ISO/IEC TR 24772–10  
Notes on this document

Effective 23 November 2020, this document is being moved to “github”. Contact Stephen.michell@maurya.on.ca to gain access.

This document is a draft of a Guidance to avoiding programming language vulnerabilities in C++.

At this point in time, the following clauses are essentially completed first pass.

* 6.2 type system
* 6.3 Bit representation
* 6.4 Floating Point
* 6.5 Enumerator issues \[CCB\],
* 6.6 Conversion errors
* 6.7 String termination
* 6.8 Buffer boundary violation
* 6.9 Unchecked array indexing
* 6.10 Unchecked array copying (needs to be revisited)
* 6.11 Pointer type conversions
* 6.12 Pointer arithmetic
* 6.13 Null pointer dereference \[XYH\],
* 6.14 Dangling reference to heap
* 6.15 Arithmetic wrap-around error
* 6.16 Using shift operations for multiplication and division
* 6.17 Choice of clear names \[NAI\]
* 6.18 Dead Store
* 6.19 Unused variables
* 6.20 Identifier name reuse
* 6.21 Namespace Issues
* 6.22 Initialization of variables \[LAV\]
* 6.23 Operator precedence and associativity
* 6.24 Side effects and order of evaluation
* 6.25 Likely incorrect expression
* 6.26 Dead store,
* 6.27 Switch statements and static analysis
* 6.28 Demarcation of control flow
* 6.29 Loop control variables
* 6.30 Off-by-one errors
* 6.31 Structured programming
* 6.32 Passing parameters and return values
* 6.33 Dangling references to stack frames
* 6.34 Subprogram signature mismatch
* 6.35 Recursion
* 6.36 Ignored error status and unhandled exceptions
* 6.37 Type breaking reinterpretation of data
* 6.38 Deep vs shallow copying \[YAN\]
* 6.39 Memory leak and heap fragmentation
* 6.41 Inheritance
* 6.42 Violations of the Liskov substitution principle
* 6.43 Redispatching
* 6.44 Polymorphic variables
* 6.45 Extra intrinsics
* 6.46 Argument passing to library functions
* 6.47 Inter-language calling
* 6.48 Dynamically-linked code and self-modifying code \[NYY\]
* 6.49 Library Signature
* 6.50 Unanticipated exceptions from library routines
* 6.51 Pre-processor directives
* 6.52 Suppression of language-defined run-time checking
* 6.53 Provision of inherently unsafe operations
* 6.54 Obscure language features
* 6.55 Unspecified behaviour
* 6.56 Undefined behaviour
* 6.57 Implementation-defined behaviour
* 6.58 Deprecated language features
* 6.59 Concurrency \-- Activation
* 6.60 Concurrency -- Directed termination
* 6.64 Uncontrolled format string

TBD

* 6.2 Type system -- issues being fed from 6.40 and elsewhere
* 6.61 Concurrent data access
* 6.62 Concurrency -- Premature termination
* 6.63 Protocol lock errors

Participants at meeting 23 November 2020

Stephen Michell

Paul Preney

Peter Sommerlad

Richard Corden

Erhard Ploedereder

Clive Pygott

Michael Wong

Edition 1

ISO/IEC JTC 1/SC 22/WG 23

Secretariat: ANSI

Information Technology — Programming languages — Guidance to avoiding vulnerabilities in programming languages – Part 10 – Vulnerability descriptions for the programming language C++

Document type: International standard

Document subtype: if applicable

Document stage: (10) development stage

Document language: E

*Élément introductif — Élément principal — Partie n: Titre de la partie*

Warning

This document is not an ISO International Standard. It is distributed for review and comment. It is subject to change without notice and may not be referred to as an International Standard.

Recipients of this draft are invited to submit, with their comments, notification of any relevant patent rights of which they are aware and to provide supporting documentation.

**Copyright notice**

This ISO document is a working draft or committee draft and is copyright-protected by ISO. While the reproduction of working drafts or committee drafts in any form for use by participants in the ISO standards development process is permitted without prior permission from ISO, neither this document nor any extract from it may be reproduced, stored or transmitted in any form for any other purpose without prior written permission from ISO.

Requests for permission to reproduce this document for the purpose of selling it should be addressed as shown below or to ISO’s member body in the country of the requester:

*ISO copyright office*

*Case postale 56, CH-1211 Geneva 20*

*Tel. + 41 22 749 01 11*

*Fax + 41 22 749 09 47*

*E-mail copyright@iso.org*

*Web www.iso.org*

Reproduction for sales purposes may be subject to royalty payments or a licensing agreement.

Violators may be prosecuted.

Contents Page

[Foreword vii](#foreword)

[Introduction viii](#introduction)

[1. Scope 1](#scope)

[2. Normative references 1](#normative-references)

[3. Terms and definitions, symbols and conventions 1](#X4d9250214786b01157a0f16c928f0527361ad20)

[3.1 Terms and definitions 1](#terms-and-definitions)

[4. Language concepts 4](#language-concepts)

[5. Avoiding programming language vulnerabilities in C++ 4](#section)

[6. Specific Guidance for C++ Vulnerabilities 6](#specific-guidance-for-c-vulnerabilities)

[6.1 General 6](#general)

[6.2 Type System [IHN] 6](#type-system-ihn)

[6.3 Bit Representations [STR] 7](#bit-representations-str)

[6.4 Floating-point Arithmetic [PLF] 8](#floating-point-arithmetic-plf)

[6.5 Enumerator Issues [CCB] 8](#enumerator-issues-ccb)

[6.6 Conversion Errors [FLC] 9](#X9cfda3cbe4730cd716a6d31f12e737a778b7ef7)

[6.7 String Termination [CJM] 10](#string-termination-cjm)

[• Use std::string or similar, in preference to C-style arrays of chars 11](#X84aecf3c753348107c666965525eefe0891b324)

[6.8 Buffer Boundary Violation [HCB] 11](#buffer-boundary-violation-hcb)

[6.9 Unchecked Array Indexing [XYZ] 12](#unchecked-array-indexing-xyz)

[6.10 Unchecked Array Copying [XYW] 13](#unchecked-array-copying-xyw)

[6.11 Pointer Type Conversions [HFC] 13](#pointer-type-conversions-hfc)

[6.12 Pointer Arithmetic [RVG] 15](#pointer-arithmetic-rvg)

[6.13 NULL Pointer Dereference [XYH] 16](#null-pointer-dereference-xyh)

[6.14 Dangling Reference to Heap [XYK] 16](#dangling-reference-to-heap-xyk)

[6.15 Arithmetic Wrap-around Error [FIF] 18](#arithmetic-wrap-around-error-fif)

[6.16 Using Shift Operations for Multiplication and Division [PIK] 18](#X5f56d22c3626f3f512756efb6c763134eb6f9b6)

[6.17 Choice of Clear Names [NAI] 19](#choice-of-clear-names-nai)

[6.18 Dead Store [WXQ] 19](#dead-store-wxq)

[6.19 Unused Variable [YZS] 20](#unused-variable-yzs)

[6.20 Identifier Name Reuse [YOW] 20](#identifier-name-reuse-yow)

[6.21 Namespace Issues [BJL] 21](#Xe6608fecd4b6b786bda30fa6ba97a6420d76329)

[6.22 Initialization of Variables [LAV] 21](#initialization-of-variables-lav)

[6.23 Operator Precedence and Associativity [JCW] 21](#Xeb4b392fc60d52e0e62d52a00eabaa492eaa9e0)

[6.24 Side-effects and Order of Evaluation of Operands [SAM] 21](#X187ad205340e16f022c4e8cacaf0507e2964127)

[6.25 Likely Incorrect Expression [KOA] 22](#likely-incorrect-expression-koa)

[6.26 Dead and Deactivated Code [XYQ] 24](#dead-and-deactivated-code-xyq)

[6.27 Switch Statements and Static Analysis [CLL] 24](#Xc86cceedfcf0c77c094b3df762cc0f756518ffc)

[6.28 Demarcation of Control Flow [EOJ] 25](#demarcation-of-control-flow-eoj)

[6.29 Loop Control Variables [TEX] 26](#loop-control-variables-tex)

[6.30 Off-by-one Error [XZH] 27](#off-by-one-error-xzh)

[6.31 Structured Programming [EWD] 28](#structured-programming-ewd)

[6.32 Passing Parameters and Return Values [CSJ] 28](#passing-parameters-and-return-values-csj)

[6.33 Dangling References to Stack Frames [DCM] 29](#dangling-references-to-stack-frames-dcm)

[6.34 Subprogram Signature Mismatch [OTR] 30](#subprogram-signature-mismatch-otr)

[6.35 Recursion [GDL] 31](#recursion-gdl)

[6.36 Ignored Error Status and Unhandled Exceptions [OYB] 31](#X7b5801efcad7d4f25e284b0270c97edc564ec06)

[6.37 Type-breaking Reinterpretation of Data [AMV] 32](#X3c61d4dde3e544e3427c66a2e5f6e053f7545a4)

[6.38 Deep vs. Shallow Copying [YAN] 33](#deep-vs.-shallow-copying-yan)

[6.39 Memory Leak and Heap Fragmentation [XYL] 33](#memory-leak-and-heap-fragmentation-xyl)

[6.40 Templates and Generics [SYM] 34](#templates-and-generics-sym)

[6.41 Inheritance [RIP] 35](#inheritance-rip)

[6.41.1 Applicability to language 35](#applicability-to-language-39)

[6.41.2 Guidance to language users 37](#guidance-to-language-users-37)

[6.42 Violations of the Liskov Substitution Principle or the Contract Model [BLP] 37](#X905dbd3391b2a038bbf387b2b00b36e59d5f153)

[6.42.1 Applicability to language 37](#applicability-to-language-40)

[6.42.2 Guidance to language users 38](#guidance-to-language-users-38)

[6.43 Redispatching [PPH] 38](#redispatching-pph)

[6.43.1 Applicability to language 38](#applicability-to-language-41)

[6.43.2 Guidance to language users 39](#guidance-to-language-users-39)

[6.44 Polymorphic variables [BKK] 39](#polymorphic-variables-bkk)

[6.44.1 Applicability to language 40](#applicability-to-language-42)

[6.44.2 Guidance to language users 41](#guidance-to-language-users-40)

[6.45 Extra Intrinsics [LRM] 42](#extra-intrinsics-lrm)

[6.46 Argument Passing to Library Functions [TRJ] 42](#Xc417c172da80bb70cea1f9e7af9871a63840609)

[6.47 Inter-language Calling [DJS] 42](#inter-language-calling-djs)

[6.48 Dynamically-linked Code and Self-modifying Code [NYY] 45](#Xfd4a54be17d3eceb94ec5c5981ef893e36604c5)

[6.49 Library Signature [NSQ] 46](#library-signature-nsq)

[6.50 Unanticipated Exceptions from Library Routines [HJW] 47](#X165926c342ab175b2667683b6cb7551fc7b18d8)

[6.51 Pre-processor Directives [NMP] 48](#pre-processor-directives-nmp)

[6.52 Suppression of Language-defined Run-time Checking [MXB] 49](#Xe4ed1f6d7e7049a22cf11fb58e9b78561722b0b)

[6.53 Provision of Inherently Unsafe Operations [SKL] 49](#X2e6df9becdc68c65fdb33865090fafdb8a33550)

[6.54 Obscure Language Features [BRS] 49](#obscure-language-features-brs)

[6.55 Unspecified Behaviour [BQF] 50](#unspecified-behaviour-bqf)

[6.56 Undefined Behaviour [EWF] 50](#undefined-behaviour-ewf)

[6.57 Implementation–defined Behaviour [FAB] 51](#implementationdefined-behaviour-fab)

[6.58 Deprecated Language Features [MEM] 52](#deprecated-language-features-mem)

[6.59 Concurrency – Activation [CGA] 52](#concurrency-activation-cga)

[6.60 Concurrency – Directed termination [CGT] 52](#concurrency-directed-termination-cgt)

[6.60.1 Applicability to language 53](#X70825a5f13fa91411795bc5a5128191ea1b0e8f)

[6.60.2 Guidance to language users 53](#guidance-to-language-users-55)

[6.61 Concurrent Data Access [CGX] 53](#X326b0b69f0e6556e79033b7e9c0299c089005a3)

[6.62 Concurrency – Premature Termination [CGS] 53](#concurrency-premature-termination-cgs)

[6.63 Protocol Lock Errors [CGM] 53](#protocol-lock-errors-cgm)

[6.64 Uncontrolled Format String [SHL] 54](#uncontrolled-format-string-shl)

[7. Language specific vulnerabilities for C 54](#language-specific-vulnerabilities-for-c)

[8. Implications for standardization 54](#implications-for-standardization)

[Bibliography 57](#bibliography)

[Index 60](#index)

# Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of the joint technical committee is to prepare International Standards. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

In exceptional circumstances, when the joint technical committee has collected data of a different kind from that which is normally published as an International Standard (“state of the art”, for example), it may decide to publish a Technical Report. A Technical Report is entirely informative in nature and shall be subject to review every five years in the same manner as an International Standard.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

ISO/IEC TR 24772-10, was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 22, *Programming languages, their environments and system software interfaces*.

# Introduction

This Technical Report provides guidance for the programming language C++, so that application developers using or considering C++ will be better able to avoid the programming constructs that lead to vulnerabilities in software written in the C++ language and their attendant consequences. This guidance can also be used by developers to select source code evaluation tools that can discover and eliminate some constructs that could lead to vulnerabilities in their software. This report can also be used in comparison with companion Technical Reports and with the language-independent report, TR 24772–1, to select a programming language that provides the appropriate level of confidence that anticipated problems can be avoided.

This technical report part is intended to be used with TR 24772–1, which discusses programming language vulnerabilities in a language independent fashion. It is also intended to be used with TR 24772-3, which discusses how the vulnerabilities introduced in TR 24772-1 are manifested in C, which is a subset of C++.

It should be noted that this Technical Report is inherently incomplete. It is not possible to provide a complete list of programming language vulnerabilities because new weaknesses are discovered continually. Any such report can only describe those that have been found, characterized, and determined to have sufficient probability and consequence.

**Information Technology — Programming Languages — Guidance to avoiding vulnerabilities in programming languages — Vulnerability descriptions for the programming language C++**

# 1. Scope

This Technical Report specifies software programming language vulnerabilities to be avoided in the development of systems where assured behaviour is required for security, safety, mission-critical and business-critical software. In general, this guidance is applicable to the software developed, reviewed, or maintained for any application.

Vulnerabilities described in this Technical Report document the way that the vulnerability described in the language-independent TR 24772–1 are manifested in C++.

# 2. Normative references

The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

ISO/IEC 14882:2014 — *Programming Languages—C* ++

ISO/IEC TR24772–3 -- Information Technology — Programming Languages — Guidance to avoiding vulnerabilities in programming languages — Vulnerability descriptions for the programming language C

# 3. Terms and definitions, symbols and conventions

## 3.1 Terms and definitions

For the purposes of this document, the terms and definitions given in ISO/IEC 2382, in TR 24772–1, in 14882:2014 and the following apply. Other terms are defined where they appear in *italic* type.

The following terms are in alphabetical order, with general topics referencing the relevant specific terms.

3.1.1

TBD

3.1.2

access:

An execution-time action, to read or modify the value of an object.

Note 1: Where only one of two actions is meant, read or modify. Modify includes the case where the new value being stored is the same as the previous value. Expressions that are not evaluated do not access objects

**3.1.3**

access protection

**ADL**

argument dependent lookup

**alignment**  
requirement that objects of a particular type be located on storage boundaries with addresses that are particular multiples of a byte address

**3.1.3**

**argument**  
the expression in the comma-separated list bounded by the parentheses in a function call expression, or a sequence of preprocessing tokens in the comma-separated list bounded by the parentheses in a function-like macro invocation

Note 1: Also called actual argument

Note 2: An argument replaces a *formal parameter* as the call is realized.

**3.??**

argument dependent lookup

lookup that finds additional overloads from the namespaces of the types of the arguments used in unqualified function calls

**3.1.4**

**behaviour**  
an external appearance or action [recommend removal - standard comp sci term]

**3.1.5**

**bit**  
the unit of data storage in the execution environment large enough to hold an object that may have one of two values

Note: It need not be possible to express the address of each individual bit of an object [recommend removal - standard comp sci term]

**3.1.6**

**byte**  
the addressable unit of data storage large enough to hold any member of the basic character set of the execution environment

Note: It is possible to express the address of each individual byte of an object uniquely. A byte is composed of a contiguous sequence of bits, the number of which is implementation-defined. The least significant bit is called the low-order bit; the most significant bit is called the high-order bit. [recommend removal - standard comp sci term]

**3.1.7**

**character**  
abstract member of a set of elements used for the organization, control, or representation of data and ideally when treated sequentially represents text

**correctly rounded result**  
representation in the result format that is nearest in value, subject to the current rounding mode, to what the result would be given unlimited range and precision

**3.1.8**

**class**  
a user-defined type declared with the class-key ‘class’ or ‘struct’ {.ul}

**3.1.9**

**concrete**

Recommend deletion, only used in 6.40 Templates and Generics

**3.1.10**

**diagnostic message**

informational message that is either an error or warning about an issue detected by the implementation

**[3.1.11]**{.ul}\*\*

**dynamic dispatch**

[recommend removal] {.ul}

3.1.12

**encapsulation**

recommend removal - all usages in the document satisfy standard English use of encapsulate

**3.1.13**

**formal parameter**

object declared as part of a function declaration or definition that acquires a value on entry to the function, or an identifier from the comma-separated list bounded by the parentheses immediately following the macro name in a function-like macro definition.

**[3.1.xx] friend{.ul}**

function or class that can access the private and protected members of a specific class

**[3.1.xx] hidden friend{.ul}**

friend function that is only declared within a class or class template definition and hence is only found by ADL

**3.1.xx**

**Implementation**

toolchain that is used to build and support the execution of the C++ program

**3.1.15**

**implementation-defined behaviour**

behaviour, for a well-formed program construct and correct data, that depends on the implementation and that each implementation documents

**3.1.16**  
**implementation-defined value**

unspecified value where each implementation documents how the choice for the value is selected. [recommend removal - term not use other than note(s)]

**3.1.17**

**implementation limit**

restriction imposed upon programs by the implementation.

**3.1.18**

**indeterminate value**

unspecified value or a trap representation

**3.1.19**  
**indeterminately sequenced**

sequenced in a way that one of two evaluations will be executed before the other but in an unspecified order

**3.1.20**

**Inheritance**

TBD [recommend removal - well understood in OO languages]

**3.1.21**

**language type**

see block-structured language, comb-structured language (Non-responsive) [recommended removal - inadequate relevancy]

**3.1.21**

**locale-specific behaviour**

behaviour that depends on local conventions of nationality, culture, and language that each implementation documents

**3.1.22**

**memory location**

an object of scalar type or a maximal sequence of adjacent bit-fields all having nonzero width [recommend removal and move to 6.3]

**3.23**  
**multibyte character**

sequence of one or more bytes representing a member of the extended character set of either the source or the execution environment.

Note: The extended character set is a superset of the basic character set. [Recommend removal - not used]

**3.1.24**  
**namespace**  
optionally-named entity that can contain scoped declarations of any kind of entity [ensure that this is put into clause 4 on general concepts]

**3.25**

**object**

region of data storage in the execution environment, the contents of which can represent values and that can be interpreted as having a particular type

**3.1.26**

**overload**

the use of the same symbol name to denote different entities

**3.1.27**

**override**

replacing the implementation of an inheritable function in a derived class

**3.1.28**  
pointer-like-type

a set of types whose values depend upon the existence and value of another object, such as types for references, pointers, iterators and views

**3.1.29**

**{Protected]{.ul}**

visible only to itself and derived classes and friends

**3.1.30**

**private**

visible only to the class itself and friends

**3.1.31**

**Public**

visible without restriction

**3.1.33**

**recommended practice**

specification that is strongly recommended as being in keeping with the intent of the language standard, but that may be impractical for some implementations

[recommend removal - not used] \*\*3.1.34((

**runtime-constraint**

a constraint imposed on an executing program [TBD - needs refinement] [continue from here - 7 August 2023]

**3.1.35**

**single-byte character**

bit representation that fits in a byte (binary representation?)

**3.1.36**

**static**

TBD

\*\*3.1.37

**STL**

TBD

**3.1.38**

**standard library**

[TBD]

**3.1.39**

**template**

TBD

**3.1.40**

**trap representation**

object representation that need not represent a value of the object type

**3.1.41**

**undefined behaviour**

use of a non-portable or erroneous program construct or of erroneous data, for which the language standard imposes no requirements

Note: Undefined behaviour ranges from ignoring the situation completely with unpredictable results, to behaving during translation or program execution in a documented manner characteristic of the environment (with or without the issuance of a diagnostic message), to terminating a translation or execution (with the issuance of a diagnostic message). An example of, undefined behaviour is the behaviour on integer overflow.

**3.1.42**

**unspecified behaviour**

use of an unspecified value, or other behaviour where the language standard provides two or more possibilities and imposes no further requirements on which is chosen in any instance

Note: For example, unspecified behaviour is the order in which the arguments to a function are evaluated.

**3.1.43**

**unspecified value**

valid value of the relevant type where the language standard imposes no requirements on which value is chosen in any instance

Note: An unspecified value cannot be a trap representation.

**3.1.44**

**value**

precise meaning of the contents of an object when interpreted as having a specific type (specific type or specified type?)

Note: See implementation-defined value, indeterminate value, unspecified value, trap representation

**3.1.45**

**virtual**

TBD

**3.1.45**

**wide character**

bit representation capable of representing any character in the current locale

# 4. Language concepts

*This clause requires a rewrite. See C++ Core Guidelines CPL for a good explanation of the differences*.

## 4.1 Overview

***THIS REQUIRES MORE WORK***

Define unchecked (random) access in clause 3 or explain C++ approach. Likely needs a new subclause. Indexing into raw memory is a random access with no checking. In the STL, the [] operator does random access without checking. The function at was added to provide range checking, including throwing an exceptiion if the check fails.

C++ is a strongly- and statically-typed language: all variables and expressions must have a type. C++ also permits implicit and explicit conversions between types.

C++ has a rich type system with many nuances. In addition to the C base types (int, long, float, double, char, and arrays with their C-style vulnerabilities), C++ provides the following:

* Distinct types for char, signed char, unsigned char
* bool with a true Boolean context
* enum with fixed underlying type
* Classes as the primary user-defined types
* Arrays
* References in addition to pointers
* Parameterized types

Many vulnerabilities can be mitigated more easily by using library facilities rather than the base language types. (e.g. std::string rather than char\*)

* Use of the `explicit`{.cpp} keyword for constructors and conversion  
  operators
* `operator bool()`{.cpp} discussion
* many built-in implicit conversions, refer to TR 24772-3 clause 6.2  
  and other clauses (C)
* conversion to bool and null pointer conversions
* legacy code operator void\\* - change to explicit operator bool
* C-style casts break type safety.
* static\_cast
* explicit casts highlight mismatches between the design and  
  implementation.
* const and volatile
* constexpr -- needs a writeup -- (in C++:11 , encouraged heavy stack  
  use and possible exhaustion).

Narrowly tailored number-like class types, such as time\_point and duration, improve safety by providing only safe and appropriate operations. User-defined types tailored to a particular use case can provide additional safety.

C++ was initially defined as a syntactic superset of the C programming language: adding object oriented features such as classes, encapsulation, dynamic dispatch, namespaces and templates. It was a “syntactic superset” because whilst there is a core of C++ that is syntactically identical to C, it has always been the case that there are subtle semantic differences between the two, for example:

* Historically, C permitted the use of a function before its declaration (though this is now deprecated in C) . This is illegal in C++
* Where a struct is defined within another struct, in C the inner declaration is in effect made at file scope, so the definition is available for use later in the program. In C++, the inner declaration name is qualified by that of the parent, so without qualification, the inner struct cannot be used later in the program, as in the following example

struct S1 {  
 struct S2 {...} m1;  
 ...  
};  
struct S2 v1; /\\* legal in C not C++ \\*/  
S1::S2 v2 // legal in C++ not C

Subsequently, the two languages have diverged, both adding features not present in the other. Not withstanding that, there is still a significant syntactic and semantic overlap between C and C++, so the starting point for this report has been the equivalent for C. However, in many cases, the additional features of C++ provide mechanisms for avoiding the vulnerabilities inherited from C, and these are reflected in the following sections.

*Include discussions of Object orientation,* ***static****, and* ***const,*** *scoped enumerations*

## 4.2 Type System

* implicit conversions
* templates? Bring tutorial material from clause 6.2/[#IHN] Type system clause 6.6/[#FLC] Conversion errors, clause 6.11/[#HFC] Pointer type conversions, clause 6.37[#AMV] Type breaking reinterpretation of data clause 6.41[#RIP] Inheritance (includes Mix-ins and multiple inheritance) clause 6.44/[#BKK] Polymorphic variables

Mix-ins (6.20, 6.40, 6.41)

### Inheritance

C++ supports user-defined class types with one or more base classes. Dynamic polymorphism requires public bases defining virtual member functions.

C++ provides access specifiers that allow inheritance to restrict the visibility of inherited members in their subclasses.

A member declared in a class hides all members of the same name in any of its base classes. Such names can be reintroduced by the using declaration. For example:

struct Base {  
 int f(int i);  
 };  
  
 struct Derived : public Base {  
 // using Base::f;  
 int f(char c);  
 int g() {  
 return f(123); // Surprise, f(char) is called  
 }  
};

If the using declaration is uncommented above, then Base::f(int) is called.

For multiple inheritance users can request virtual inheritance, which causes data members inherited from the same ancestor class along multiple inheritance paths to be present only once; otherwise the data members are replicated and referring to them is ambiguous unless qualified by the name of the base class from which they are inherited. When members of equal name are inherited from multiple sources, C++ rejects an unqualified use of the common name as ambiguous as long as they are not hidden.

***THIS REQUIRES MORE WORK***

A particular area that is misunderstood is integral promotion. It can be confusing because promotion can result in changing the internal representation of an unsigned type to/from a signed type. For expressions formed with operands of unscoped enumeration type or integral types with a conversion rank smaller than int, *integral promotion* occurs before further implicit conversions happen. *Integral promotion* on unsigned types can convert these to the signed type int. Undefined behavior [[EWF]](#EWF) can occur due to signed integer arithmetic overflow even when the operands are of an unsigned type. Assume for example, - signed and unsigned short occupies 16 bits - signed and unsigned int occupies 32 bits then the following code causes undefined behavior:

unsigned short const x = 0xfff0;  
 x \* x; // signed integer overflow, result will not fit in signed 32 bit int

Note: C++ also uses the term promotion to apply to a subset of conversion that apply to intergral and floating point types.

Implicit, i.e., automatic, conversions to a type T can be performed, for example, in the following situations:

1. If the declaration, T t=e;, is defined for some expression, e, and some invented variable, t [C++17, Clause 7 [conv], para 3];
2. In expressions involving operands of operators (e.g., +, -, \*, /, etc.) subject to the requirements of each operators' operands [C++20, Clause 7.3 [conv], para 2.1];
3. For example, the expression, 5 + 6.5, has operands of type int and double. Per language rules, the integer operand will be implicitly converted to double, i.e., the expression becomes double(5) + 6.5, i.e., 5.0 + 6.5.
4. In boolean contexts, such as
   * operands of logical operators,
   * the condition of the conditional operator, and
   * the condition of the if-statement or loop-statements, contextual conversion to bool implicitly happens from arithmetic types, pointers, and classes with a possibly explicit operator bool(). [C++17, Clause 7 [conv], para 4];
5. In the expression of a switch statement: the implicit conversion will be to an integral type [C++17, Clause 7 [conv], para 2.3];
6. In an expression that initializes an object (e.g., an argument to a function call, the expression in a return statement) [C++17, Clause 7 [conv], para 2.4];
7. When a non-explicit class/struct/union constructor can be invoked on an object resulting in some desired type, T, from initial objects passed to the constructor; and
8. When a conversion operator has not been declared explicit, it can be implicitly invoked on an object resulting in some desired type, T, from an initial type.

## 4.2 Symbol Lookup and Overload Resolution

***THIS REQUIRES MORE WORK***

* special members
* templates?

scopes, names, ADL, using

Add to clause 4 “Language concepts” an issue on C++ symbol lookup issues considering the following: 1. Minimize the set of names that are available to avoid referring to an item that was not the intended target 2. hidden friends 3. ADL (argument dependent lookup) 4. Lack of this-> in class templates 5. Minimizing lexical scope and visibility 6. Minimizing names in global scopes 7. Minimizing use of “using” directives

namespace NS {  
template <typename T>  
struct A  
{  
 template <typename S, typename Q> friend void add (S, Q);  
};  
}  
  
struct B {  
 B(int);  
 B (NS::A<int> const &);  
};  
  
B & add(B const & lhs, B const & rhs);  
  
void bar (NS::A<int> & a, B const & b)  
{  
 add(a, b);  
}

A C++ program may span multiple scopes, and when a function is called, the compiler may have many choices to select from to resolve the actual function that will be called. The set of names found for a function depend upon whether the name is qualified or if it is unqualified.

Unqualified lookup begins the search in the current scope and works its way out to the global scope and stops when it finds the first matching name. The search will include: - Names directly visible; - (Non-dependent) base classes if the scope is for a member; - Names introduced by the using declaration; - Names introduced by the using directive; and - Members of inline namespaces After these steps, Argument-Dependent Lookup (ADL) occurs. In ADL, the scope(s) of type of the function’s arguments are also searched. This search also includes friends that would not normally be found. ADL never finds member functions.

Unqualified name lookup in templates has two phases: - When the template is initially parsed, names are looked up as above, excluding dependent bases; and - When the template is instantiated, ADL is performed for function names, but ADL does not find member functions.

A qualified name lookup takes place when: - the name includes its scope (e.g a::foo); or - the name is used in a class member access expression (this->foo). Qualified lookup only searches: - Names directly visible; - (Non-dependent) base classes if the scope is for a member; - Names introduced by the using declaration; - Names introduced by the using directive; - Members of inline namespaces; and - Names that can be resolved when a template is instantiated.

Overload resolution

## 4.4 Object Lifetime

In principle, the lifetime of an object begins immediately after a constructor completes creating the object and ends immediately before a destructor is called. C++ distinguishes references from objects. A reference has a lifetime that begins immediately after it is initialized and ends when the scope containing it completes. If the lifetime of an object designated by a reference ends before the lifetime of the reference ends, the reference is dangling and further access to its referent is undefined behaviour.

The lifetime of an object relates to its storage. There are four types of storage durations: - static - includes global and static variables; lifetime guaranteed to be the execution duration of main() (or the equivalent for freestanding implementations) - thread - variables have a lifetime of no longer than the lifetime of the containing thread - automatic - scope of declaration defines lifetime - dynamic - storage for objects is acquired from dynamic memory allocation and its lifetime is manually controlled.

The lifetime of an object can be prematurely terminated by - calling its destructor - releasing or reusing its storage - reusing its storage in a placement new expression

Accessing an object outside of its lifetime is undefined behaviour.

Sharing a reference to an object can result in incorrect behaviours if an object is terminated using one reference and later accessed via another reference. A memory leak occurs when an object who’s lifetime has not ended but there are no more variables, pointers or references to that object.

TO Do: verify that this clause identifies issues raised in: 6.11.PointerTypeConversion-HFC.md 6.13.NULLPointerDereference-XYH.md 6.14.DanglingReferenceToHeap-XYK.md 6.22.InitializationOfVariables-LAV.md 6.33.DanglingReferencesToStackFrames-DCM.md 6.38.DeepVsShallowCopying-YAN.md 6.39.MemoryLeakAndHeapFragmentation-XYL.md 6.61.ConcurrentDataAccess-CGX.md 6.63.ProtocolLockErrors-CGM.md 6.65.ModifyingConstants-UJO.md

## 4.5 Initialization

C++ provides a number of ways that an object can be initialized - Value initialization, e.g. std::string s{}; - Direct initialization, e.g. std::string s(“hello”); - Copy initialization, e.g. std::string s = “hello”; - List initialization, e.g. std::string s{‘a’, ‘b’, ‘c’}; - Aggregate initialization, e.g. char a[3] = {‘a’, ‘b’}; - Reference initialization, e.g. char& c = a[0]; - Zero initialization - Default initialization copy, move, default constructor

C++ has many forms of initilization, which generally guarantee that subsequent access to the declared object will be well-defined, however C++ does not always guarantee that it contains a legal value unless the programmer explicitly initializes it with a value. See ##6.22 Missing initialization of variables [LAV]#LAV.

The kind of initialization that happens in C++ when an explicit initializer is not used depends on the context of the declaration.

int i; // zero-initialized in global name space,   
 void foo()  
 {  
 int i; // default initialized, inderterminate value  
 int j{}; // value-initialized to zero  
 }

Non-local variables with static storage duration that are dynamically initialized can cause undefined behavior [[EWF]](#EWF) if the initialization depends on other such variables, see ##6.22 Missing initialization of variables [LAV]#LAV.

The lifetime of most objects of a class type begins when the object’s constructor has completed. In some situations binding a reference to a temporary will extend the lifetime of the temporary. See ##4.4 Lifetime and ## 6.33 Dangling References to Stack Frames [DCM]. // In any case, an object should not be accessed until its initialization is complete (including concurrency???). Richard C thinks that https://eel.is/c++draft/basic.start.dynamic#5 means that there is no danger of one thread accessing a dynamically initialized global variable before the initialization has taken place - is that the case?

When a function is called, each parameter is initialized with its corresponding argument. The initialization of a parameter, including every associated value computation and side effect, is indeterminately sequenced with respect to that of any other parameter, see ## 6.24 Side-effects and Order of Evaluation of Operands [SAM]. On the other hand, the value computation and side effects of the initializer-clauses in an initializer-list are evaluated in the order they appear.

There are many ways for a user to construct, or initialize, an object.

# 5. Avoiding programming language vulnerabilities in C++

In addition to the generic programming rules from ISO/IECTR 24772-1 clause 5.4, additional rules from this section apply specifically to the C++ programming language. The recommendations of this section are restatements of recommendations from clause 6, but represent ones stated frequently, or that are considered as particularly noteworthy by the authors. Clause 6 of this document contains the full set of recommendations, as well as explanations of the problems that led to the recommendations made.

Every guidance provided in this section, and in the corresponding Part section, is supported by material in Clause 6 of this document, as well as other important recommendations.

***TBD***

|  |  |  |
| --- | --- | --- |
| Index |  | Reference[]{custom- style=“annotation reference”} |
| 1 |  |  |
| 2 |  |  |
| 3 |  |  |
| 4 |  |  |
| 5 |  |  |
| 6 |  |  |

Need to consider C++-11, 14 and 17.

# 6. Specific Guidance for C++ Vulnerabilities

## 6.1 General

This clause contains specific advice for C++ about the possible presence of vulnerabilities as described in TR 24772-1, and provides specific guidance on how to avoid them in C++ code. This section mirrors TR 24772-1 clause 6 in that the vulnerability “Type System [IHN]” is found in 6.2 of TR 24772–1, and C++ specific guidance is found in clause 6.2 and subclauses in this TR.

As part of its design (and with few exceptions), C++ has a common subset with the complete C language. For code portions written in the common subset, the vulnerabilities described and the advice given in ISO/IEC TR 24772-3:2020, *Part 3 – Vulnerability descriptions for the programming language C*, apply, except when this document provides refined advice. The following subclauses usually do not further acknowledge the issues from the subset since those have been adequately addressed in the referenced document. However, C++ provides mechanisms to mitigate many of the problems that arise. Please refer to the respective clauses of this document for these mitigations and related guidelines.

## 6.2 Type System [IHN]

### 6.2.1 Applicability to language

C++ is a statically typed language. In some ways, C++ is both strongly and weakly typed, as it requires all objects/expressions to have a type, but allows for some implicit conversions of values from one type to another type. The following cases require special consideration:

* Most notably are the implicit conversions between arithmetic types (including bool) that silently might truncate values or have loss of precision (for details see [Conversion Errors [FLC]](#FLC)). This shared vulnerability includes *integral promotion* from small (unsigned) integer types (e.g. unsigned char) to the signed integer type int that can lead to operations with undefined behavior [[EWF]](#EWF), where unsigned arithmetic is defined with modulo semantics.
* The standard types used to represent text characters ( char, char8\_t, wchar\_t, char16\_t, char32\_t) are treated as arithmetic types even when used to represent text and thus are subject to implicit conversions to and from other arithmetic types. Using values of character types in arithmetic expressions, especially together with other types can be confusing. Furthermore, using values of such types or sequences of such values with their corresponding string types with relational operators might not provide an expected sorting order, depending on the language or the encoding the text represents. Libraries for text processing might take such locale-specific or text-representation-specific behavior into account, whereas the built-in operators of the language don’t.
* C++ considers implicit conversions during function overload resolution, when there is not a direct match of argument type with parameter type. Sometimes such conversions lead to ambiguities, which result in a compile error. But there can also be a single best matching overload that does not correspond to developer’s expectation including conversion of argument types (see [Identifier Name Reuse [YOW]](#YOW)). Even where the code presently corresponds to programmer expectation, a later addition of code can create a new “better match”, potentially resulting in a silent change of behaviour (see [Namespace issues [BJL]](#BJL), and [Conversion Errors [FLC]](#FLC)).
* Values of an unscoped enumeration convert to their underlying integral type in arithmetic expressions (see [Enumerator Issues [CCB]](#CCB)).
* Built-in array types as well as function types will decay to a pointer type. For array types this is problematic, because the array’s extent is lost (see [Unchecked Array Indexing [XYZ]](#XYZ) and [Pointer Arithmetic [RVG]](#RVG))
* Pointer types implicitly convert to pointer-to-void (see [Pointer Type Conversions [HFC]](#HFC)).
* Pointer and reference types referring a derived class object can implicitly convert to a pointer or reference to one of its public base classes respectively (see [Polymorphic Variables [BKK]](#BKK)).
* User-defined class types can define conversion operators and converting constructors that can be called with a single argument. If these member functions are not declared with the keyword explicit they are eligible for use in implicit conversions.
* Implicit conversions from an arithmetic type to bool often indicate a missing comparison. Conversions from bool in arithmetic expression can obfuscate code. For example, the following code relies on false converting to/from zero and true to one and from non-zero:

double fluxcompensation(double flux, bool compensate){  
 if (flux) { // double to bool conversion  
 double delta = compute\_delta();  
 double const compensate\_v = 1.4;  
 return flux + delta \* compensate; // bool to double conversion  
 }   
 return 1.;  
}

* In a boolean context, conversions that would not occur implicily can take place, specifically, a pointer-to-bool conversion and the application of an explicit operator bool() conversion operator. These conversions are idiomatic C++ in boolean contexts; however, some prefer explicit tests to show intent, for example, if (p) { \*p = 42; } vs. if (p != nullptr) { \*p = 42; }.

Note that type aliases (using, typedef) do not define a different type from their alias just a different name and thus do not incur any conversion between the alias and the aliased type.

Instead of using the built-in arithmetic types or generic library types such as std::string for your domain values, C++ allows to wrap them in user-defined-class types as so-called *strong types*. For integral values, enum class types can also be used. Strong types provide only those operator overloads and conversions for each such type that make sense in the application domain. User-defined-literal operators help with providing constants of appropriate strong types. Such strong types provide full control of conversions and operations available, avoiding semantically unsound operations that the built-in or other generic types might provide.

For example, a very simple strong type representation of temperature values can be implemented as follows:

struct Celsius {  
 double value;  
};  
struct Fahrenheit {  
 double value;  
};  
Fahrenheit convert\_to\_fahrenheit(Celsius c){  
 return { 9\*c.value/5+32};  
}  
//...  
Celsius wrong = convert\_to\_fahrenheit({20.}); // doesn't compile

In a realistic scenario using a library for strong type support eases the definition and use of strong types.

### 6.2.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects, C++ software developers can: - Use the avoidance mechanisms of ISO/IEC TR 24772-1:2019, 6.2.5. and the guidance provided in the different related sections of this document.

* Be aware of the rules of the type system, overload resolution, and implicit conversions to avoid vulnerabilities.
* Enable compiler warnings regarding implicit conversions and/or use static analysis tools that provide such warnings.
* Define any constructor of a class that can be called with a single argument of a different type as explicit.
* Define any conversion operator as explicit
* Use strong types for domain values instead of the built-in types. On system boundaries, e.g., for input, convert a read value immediately to the appropriate strong type.
* When defining variables of arithmetic type, use a braced-initializer to prevent a potential narrowing conversion from the initial value’s type.
* Avoid the use of the following text-representing character types, char, char8\_t, wchar\_t, char16\_t and char32\_t, in arithmetic or relational expressions. Use an appropriate text processing library instead for character classification, conversion, comparison, and further processing.
* Use a consistent style in a project with respect to implicit pointer conversions in boolean contexts.

## 6.3 Bit Representations [STR]

### 6.3.1 Applicability to language

This vulnerabilities described in ISO/IEC 24772-1:2024 clause 6.3 is applicable to C++. The “endianness” of integer types and packing of bit fields are implementation-defined properties and not portable.

The standard library type std::endian allows to portably check the endianness of a platform and code can use this information to operate on individual bytes of a machine word in the correct order.

There is no portable mapping from bitfields in a struct to individual bits in a machine word. Therefore, C++ bitfields should not be used to directly map to bits in hardware, even though the compiler provides suitable mapping and manipulation operations. A further complication is that accessing a bitfield can often not easily be performed atomically, because the non-participating bits of a memory location need to be read before the relevant bits can be mutated through masking, and the whole memory location has to be written again. It is possible to simulate bitfields with a defined layout through library class types that implement the required masking operations.

For individual bits std::bitset<N> and std::vector<bool> can provide suitable representations at run time, but don’t support a direct mapping to machine words. However, be aware that std::vector<bool> does not in general behave like a std::vector which can cause generic code to misbehave.

C++ provides a rich set of bitwise operators that can be used to address the issues of bit manipulation in a portable way. However, the shift operation can result in undefined behavior [[EWF]](#EWF) when shifting by a negative or too large value, or when shifting a signed operand. It is advisable to use bit operations only on appropriate unsigned integral types with a known width while being careful of potential integral promotion that might change a small unsigned operand type to be promoted to a signed integer type. When bitwise memory operations are needed, it is good practice to encapsulate such operations in a class type’s member functions.

For representing individual bitmasks values employed in bit operations, it is advisable to put the corresponding named constants in an enumeration type with the appropriate underlying type for easier recall.

While a bit shift of an integral value can be viewed as a multiplication or division by a power of two, it should not be used in arithmetic expression to implement such an operation. Compilers will automatically implement such a multiplication in the most efficient way, there is no need to obfuscate multiplication and division as shift operations.

Except for specific situations (trivial types), objects of class type can not be assumed to have a layout appropriate to be manipulated on a byte or bitwise level. Depending on the size and alignment of its data members a class type might have padding bytes between members. The absence of padding in a trivially copyable type T can be checked with a static\_assert(std::has\_unique\_object\_representations\_v<T>).

std::bit\_cast can be used to reinterpret suitably sized trivial types on a bitwise level, i.e., for accessing the binary representation of a floating point value. However, for types with padding bits that do do not participate in an object’s value representation, the corresponding bits in a bit\_cast result have indeterminate values. If those bits are used to compare for equality, with the function memcmp for instance, the padding bits may differ and cause false negatives.

Malicious code could use such padding bits as a *secret channel* which might be accessed through copying the underlying bytes.

See C++ Core Guidelines ES101 use unsigned types for bit manipulation.

### 6.3.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects, C++ software developers can:

* Follow the avoidance mechanisms of ISO/IEC 24772-3 clause 6.3.2
* Avoid the use of shift operations to implement multiplication or division by powers of two.
* Use built-in bitwise operations only with operands of unsigned integral types or enums with an unsigned underlying type.
* When performing bitwise operations on operands of an unsigned type that gets promoted to a signed integral type, cast the result to the corresponding unsigned type immediately.
* Ensure that the right-hand operand of a shift operation is not negative or greater than the number of bits the left-hand operand. Use static analysis tools or other means to prove that the right hand operand of the shift is not out of bounds.
* Prefer std::bit\_cast over other means such as reinterpret\_cast to access the binary representation of an object.
* Prefer abstractions in a class type encapsulating bit manipulations using unsigned types of well-defined width, such as std::bitset, over non-portable bitfields.
* Forbid the use of std::memcmp, std::memcpy, or std::memmove on objects of type T unless std::has\_unique\_object\_representations\_v<T> is true.

## 6.4 Floating-point Arithmetic [PLF]

### 6.4.1 Applicability to language

The vulnerability as described in ISO/IEC TR 24772-1 clause 6.4 is applicable to C++.

The C++ standard assumes IEC 60559 if std::numeric\_limits\<T>::is\_iec559 is true for the types in use. In the absence of this, C++ makes few guarantees about the behaviour of floating point numbers. In particular std::less is not a total order; std::equal is not equivalent to substitutability (NaNs compare unequal to themselves, but neither less nor greater, and negative zero compares equal to positive zero).

Sorting floating point numbers with the built-in operators violates the preconditions of sorting predicates in the presence of NaN values and may raise floating point errors. The default sorting predicate std::less is suspect to this precondition violation, resulting in undefined behavior [[EWF]](#EWF) when sorting a range of floating point values that contains NaNs.

### 6.4.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects, C++ software developers can:

* Follow the avoidance mechanisms of ISO/IEC 24772-1 clause 6.4.5
* Verify compliance to ISO/IEC/IEEE 60559:2011 at compile time through std::numeric\_limits<T>::is\_iec559. Other numeric characteristics such as min(), max(), existence of NaNs, has\_denorm, and infinities can be determined in this class template.
* Be aware that the default comparisons in the standard library may produce wrong results when used on floating point members.

## 6.5 Enumerator Issues [CCB]

### 6.5.1 Applicability to language

The vulnerability documented in ISO IEC 24772-1 clause 6.5 applies to C++, as C++ allows the specification of the values underlying the enumerators.

C++ provides scoped (enum class) and unscoped (enum) enumeration types, where an underlying integral type can be specified. Unscoped enumeration types without an *enum-base* have a non-fixed underlying type that only guarantees values in the range of the provided enumerators are valid. The latter can cause non-representable values to be assigned to a variable of such an unscoped enumeration type. For enumeration types with a fixed underlying type all values of the underlying integral type are valid. See ISO IEC 24772-3 for vulnerabilities associated with unscoped enums with no fixed underlying type, except an explicit conversion is required to convert an integer to an enumeration type object.

Casting a value to an enumeration type is undefined behavior [[EWF]](#EWF) unless the source value is within the range of values of an enumeration type. See CERT INT50-CPP. (remove CERT reference???)

enum Color {red=1, green, blue};  
short i = red; // implicit conversion  
Color g { Color(green + blue) }; // Undefined-defined behaviour since 5 will not fit into  
 // the smallest bitfield to hold red, green and blue

C++ does not support implicit conversion of a scoped enum to other types, hence, operations (such as arithmetic operations) must be explicitly provided.

In the following example, based upon the above definition,

void foo () {   
 i + red ; // well-formed for unscoped enum, ill-formed for scoped enum.  
}

Scoped enums, on the other hand, require that the conversions to/from the enumeration type must be explicit, as shown below:

enum class Color : short { red, green, blue };  
short i = static\_cast<short>(Color::red);  
  
Color operator+(Color a, short b)  
{  
 return static\_cast<Color>((static\_cast<short>(a) + b) % 3);  
}

Unscoped enumeration types implicitly promote to an integral type and can be used as the index of an array without a cast. Without a user-specified mapping to an underlying representation, there will be “holes” as documented in ISO IEC 24772-1:2024 clause 6.5 and ISO IEC TR 24772-3 clause 6.5.

### 6.5.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects, C++ software developers can:

* Prefer *scoped enumerations* with an explicit fixed underlying type.
  + See CPP Core Guidelines Enum.3 "Prefer class enums over 'plain'
  + enums".
  + See AUTOSAR A7-2-3 "Enumerations shall be declared as scoped
  + enum classes"
  + See MISRA C++ 8.5.3
  + See MISRA C++ 28.5.5
* If *unscoped enumerations* are used, follow the general advice of TR 24772-3 clause 6.5.2 as well as the following:
  + For an enumeration type with a fixed underlying type, avoid assigning integer values that are outside the range of the enumeration’s underlying type.
  + For an enumeration type with no fixed underlying type, avoid assigning integer values that cannot be represented by the enumeration type.
  + Avoid casting integer values to enumeration type values, but if unavoidable, ensure that it corresponds to a named enumerator.

## 6.6 Conversion Errors [FLC]

### 6.6.1 Applicability to language

The vulnerability as documented in ISO IEC 24772-1 clause 6.6 applies to C++. C++ includes some of the conversion mechanisms of C documented in TR 24772-3 clause 6.6.1, however C++ type conversion mechanisms differ from the mechanisms of C, as documented in ISO IEC 14882 Annex C. This subclause highlights differences where C++ provides mitigations of potential vulnerabilities found in C.

In C++, some conversions are explicit while others are implicit. Conversions can change the size of a type, whether or not the type is signed, and possibly other properties of the type. A narrowing conversion is when the target type cannot represent all the values of the original type. Many errors are associated with implicit conversions. For a comprehensive overview see clause 7.3 [conv] of [C++20]

Explicit conversions use one of the mechanisms provided by C++ through a

* type conversion operator
  + static\_cast,
  + const\_cast,
  + dynamic\_cast,
  + reinterpret\_cast),
* constructor notation with
  + parentheses (e.g., double(42)) or
  + braces (e.g., int{‘a’}) which prevents narrowing conversions, or
* a C-style cast ((int)3.14).

The C-style cast is problematic since: - It is not easily identified in code using search tools, and - The compiler checks for correctness are effectively nonexistant. In C++, a C-style cast is defined in terms of the C++ cast operators const\_cast, static\_cast, and reinterpret\_cast. In some cases, it is unspecified which cast is used, for example when a cast operation involves an incomplete type, a reinterpret\_cast may be used for the conversion which can produce an incorrect result. If the C-style cast is used instead of one of the C++-style casts, the cast expected by the programmer may not be the cast actually performed, and hence should be avoided.

Unlike C++'s other cast notations, dynamic\_cast relies on run-time type information generated by the compiler to ensure the requested conversion is valid. If it is not valid, then nullptr is returned for pointer types, otherwise an exception is thrown. [C++17, Clause 8.2.7 [expr.dynamic.cast]] Thus, dynamic\_cast is safer to use when converting down or across a hierarchy. (see [Pointer Type Conversions [HFC]](#HFC) and [Polymorphic Variables [BKK]](#BKK))

An implicit conversion to a class type can occur for a class with constructors that can be invoked with a single argument, as in the following example:

class C  
 {public:  
 C(int x=10, float y=0){...}  
 };  
  
void foo(C param){...}  
void bar( bool b){ foo(b);}

In the example above, it can be surprising that foo() is called with a boolean.

Note that this implicit conversion to a class object is the default behaviour of constructors that can be called with a single parameter. The explicit keyword can be used before the constructor to prevent this happening, as in:

explicit C(int x=10, float y=0){...}

The call foo(b) would now not be legal.

Implications of casting away const using const\_cast are described in section [Modifying Constants [UJO]](#UJO).

Other implicit conversions can sometimes result in data loss or erroneous values. This is an issue with implicit conversions since they are automatic: the programmer does not explicitly write code to do the conversion. For example, a common problem is mixing signed and unsigned integral types in arithmetic expressions. This can become a problem since the ranges of signed and unsigned integer types differ and the behaviour of signed integer arithmetic on overflow is undefined [[EWF]](#EWF) whereas unsigned integer arithmetic wraps on overflow. See subclause 4.2 for a discussion of integral promotions in C++.

The issue is not restricted to narrowing conversions, as shown below:

long l\_64 = i\_32 + i\_32; // '+' operation preformed in 32 bits  
 // widened after the operation completes (and potentially overflows).

This can be avoided by converting at least one operand to the wider type as part of the operation. Note that auto directs the compiler to use the appropriate type based on the initializer expression. Subsequent use of the auto object (such as in standard mathematical operations) can lead to implicit conversions that are not obvious in the context local to the expression. Additional problems arise as a result of implicit conversions between bool and other types, thus hiding the fact when a wrong operator is used accidentally:

auto f(unsigned i, unsigned j)  
{  
 return (i > 1) & (j = 1); // (>>, &&, ==) ?  
}

In the example above, all combinations of the corresponding operators will compile with different resulting types and results.

Similar issues arise in conversions between character types (char, char8\_t, …) and other types. Character types are provided to represent text in whatever character representation is needed.

void f(char c)  
{  
 if (c < 0) // may be always false on some platforms.  
 {}  
}

In addition to the use of *strong types* (see [Type System [IHN]](#IHN)), the implicit conversions and multitude of possible operations of integral types can be mitigated by using scoped enumeration types with the corresponding integer type as its underlying type. For example, std::byte is defined to address individual unsigned char elements (bytes) in memory without participating in arithmetic or bitwise operations.

Because C++ allows function and operator overloading, the effect of implicit conversions provides an additional mechanism of failure, by selecting an unwanted overload during overload resolution due to implicit conversions. This can influence failure modes with lookup as described in section [Namespace Issues [BJL]](#BJL). // Add overload resolution reference!!

C++ also provides a library function std::bit\_cast. This function provides the ability to preserve the bit representation when converting between unrelated types.  
If such is meaningful, then std::bit\_cast reduces the risk of some undefined behaviours compared with other type punning approaches such as casts or unions.

### 6.6.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects, C++ software developers can:

* Use the avoidance mechanisms of ISO/IEC 24772-1 clause 6.6.5:
* Avoid casts and type punning (see C++ Core guidelines ES.48), treat every cast as a candidate for refactoring. If type punning is required, use std::bit\_cast.
* Avoid narrowing conversions. (see C++ Core Guidelines ES.46)
* Prefer braces over parentheses for initialization or explicit conversions from numeric types to prevent narrowing. (see C++ Core Guidelines ES.23 and ES.64)
* Prohibit C-style casts. Use C++ type conversion operators instead, as they document intent and disallow some unsafe conversions. (see C++ Core Guidelines ES.49)
* Declare explicit
  + constructors callable with a single argument; and
  + conversion functions. (see C++ Core guidelines C.46 and C.164)
* Don’t cast away const. See subclause [Modifying Constants [UJO]](#UJO)  
  (see C++ Core Guidelines ES.50)
* Do not perform arithmetic operations on operands that can be promoted. In partcular, be aware of the arithmetic operators that promote their operands, and the types are subject to promotion.
* Avoid widening the result of an arithmetic operation; prefer instead to widen the type of one or more of the operands
* Don’t mix signed and unsigned arithmetic (see C++ Core Guidelines ES100)
* Don’t rely on implicit conversions between bool and other types.
* Don’t rely on implicit conversions between character types and other types.
* Use *strong types* (see subclause [Type System [IHN]](#IHN)) to limit implicit conversions and unwanted operations.
* When using auto ensure that you know the type that auto deduces type and resist using auto for the predefined types

## 6.7 String Termination [CJM]

### 6.7.1 Applicability to language

The vulnerability as documented in ISO/IEC TR 24772-1:2019 exists in C++ when C-style strings are used. For example, passing a non-null-terminated character array to an interface that requires a null-terminated string can result in undefined behaviour [[EWF]](#EWF). C++ provides alternative string processing capabilities that do not exhibit those vulnerabilities.

C++ provides a class template for string processing, std::basic\_string that manages the space for the string and the string length and always includes a string termination character. For example, when concatenating, the std::basic\_string object will increase in size to contain the resulting string. Futhermore, as the string is guaranteed to have a string termination character, using its underlying raw pointer as a C-style string will mitigate this vulnerability because the string termination character is present.

C++ provides the library class templates std::basic\_stringview and std::span that implement reference semantics to non-owned buffers. These types do not rely on a string termination character to determine the length of the string, thus, use of these types avoids those vulnerabilities. However, using its underlying raw pointer as a C-style string can result in these vulnerabilities because the string termination character is not guaranteed to be present.

void foo(std::string const& s, std::string\_view const& sv)  
{  
 puts(s.data()); // okay string has termination character  
 puts(sv.data()); // not okay; string termination not guaranteed  
}

### 6.7.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects, C++ software developers can:

* Avoid C-style strings.
* If using C-style strings is unavoidable, use the avoidance mechanisms of ISO/IEC 24772-3 clause 6.7.2.
* Ensure that explicit overflow checks are performed for all string operations.
* Prefer using a string type, e.g., std::string, which manages the memory of its string and handles termination in order to mitigate against this vulnerability.
* Don’t use the underlying raw pointer of a std::string\_view or a std::span with interfaces that expect C-style strings.
* Use static analysis tools to help identify occurrences of undefined behaviour.

## 6.8 Buffer Boundary Violation [HCB]

### 6.8.1 Applicability to language

The vulnerability as described in ISO/IEC TR 24772-1 clause 6.8 exists in C++ when arrays are managed using raw pointers or indexing. The range of valid raw pointers to a plain array a are from the first element to one past the last element of the array, i.e., in the range [std::begin(a)..std::end(a)). An object o can be treated as a single element array with respect to pointers referring to it.

C++ provides facilities to encapsulate code that is exposed to this vulnerability. The standard library defines features that mitigate or circumvent this vulnerability. For example, std::string, std::vector, std::deque, and iostreams manage buffers internally; using “range-for” such as for (auto &e :*some container*) and the algorithm library to access elements e of the container without the possibility of a buffer boundary violation.

However, the member function data() of the contiguous sequence containers returns a non-const pointer to the underlying elements. This allows manipulating the underlying memory directly, bypassing the safety features of the container leading to this vulnerability. For example, std::string::data() returns a non-const char\*.

When working directly with iterators referring a container, one need to ensure that those iterators are and remain valid. For example, for a container c incrementing an iterator beyond the end(c) iterator or dereferencing the iterator denoted by end(c) are undefined behavior [[EWF]](#EWF).

In general, validity of iterators requires programmer care to prevent out-of-bounds access of the underlying container:

For example, using algorithms and iterators correctly to convert an input string to lower case:

std::string to\_lowercase(std::string\_view s){  
 std::string result{};  
 transform(  
 begin(s), end(s), // input range #1  
 std::back\_inserter(result), // output iterator #2  
 [](char c){ return std::tolower(c);});  
 return result;  
}

The above example, passes two ranges of characters to the transform algorithm. Potential errors due to a boundary violation could be caused by the following changes:

* wrong input range #1, e.g., passing unrelated iterators,
* insufficient space in output range #2, e.g., passing begin(result) instead of back\_inserter(result),

The second problem occurs in the following code if the length of s is longer than 31:

std::string to\_lowercase(std::string\_view s){  
 std::string result{'\0', 31};  
 transform(  
 begin(s), end(s),   
 begin(result), // error, only space for 31 characters   
 [](char c){ return std::tolower(c);});  
 return result; // size(result) == 31  
}

An additional problem occurs when performing an operation that invalidates an in-use iterator, such as the iterator internally used by the range-for statement below:

std::string to\_lowercase(std::string s){  
 for (auto &c:s){  
 s.append(std::tolower(c)); // error, invalidates in-use iterator  
 }  
 return s;  
}

Another way that overflows can occur is through the use of C-style strings, which can be treated as arrays of characters, but mishandling of the nul termination can make overflows possible. See clause 6.7 String Termination[CJM].

Since plain (C-style) arrays when passed as function arguments decay to pointers the array dimension is lost. C++ provides several means of keeping the array dimension available to the called function:

* use std::array as parameter type,
* use std::views::counted or another view as parameter type,
* use std::span as parameter type for plain arrays,
* use std::string\_view as parameter type in favour of char const\*, or
* pass plain arrays by reference to a function template and deduce the dimension.

For further explanation and examples, see

* [SEI CTR50-CPP](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CTR50-CPP.+Guarantee+that+container+indices+and+iterators+are+within+the+valid+range)
* [SEI CTR53-CPP](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CTR53-CPP.+Use+valid+iterator+ranges)
* [SEI CTR55-CPP](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CTR55-CPP.+Do+not+use+an+additive+operator+on+an+iterator+if+the+result+would+overflow)
* [C++ core guidelines I.13](https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#i13-do-not-pass-an-array-as-a-single-pointer)

### 6.8.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects, C++ software developers can:

* Use the avoidance mechanisms of ISO/IEC 24772-1 clause 6.8.5.
* Avoid C-style arrays. If unavoidable, guidance for the use of C-style arrays is provided in TR 24772-3 clause 6.8.2.
* Avoid container functions, such as data(), that bypass the safety features of the respective containers.
* To model a fixed-size array, use a library class such as std::array.
* To model arrays with dynamically changing size, use containers of the standard library, such as std::vector or std::deque.
* Avoid using a pointer parameter or a pointer-and-size parameter pair for representing a contiguous buffer; instead use a range parameter, for example, std::views::counted, std::span, or std::string\_view.
* Prefer using range-based or iterator-based algorithms, such as those of the standard library, over pointer-manipulating or indexing loops.
* Use the range-based for loop construct to iterate within the defined bounds of a range.
* Ensure that ranges and iterators used by range-for or passed to algorithms are and remain valid.
* When performing random access by indexing, use the avoidance mechanisms of clause [6.9.2](#XYZ).
* Use static analysis tools to detect buffer boundary violations.

## 6.9 Unchecked Array Indexing [XYZ]

### 6.9.1 Applicability to language

The vulnerability as documented in ISO/IEC TR 24772-1:2022 6.9 exists in C++ when an access is performed using operator[].

C-style arrays, C-style pointers, random-access iterators, and some standard library containers allow element access via operator[] which is unchecked. However, those standard library containers also provide an access function at() that behaves like operator[], but performs a check that the access is within the bounds of the container and throws an exception otherwise.

For issues associated with exception handling and error handling, see clause [6.36](#OYB) Ignored error status and unhandled exceptions.

The parameter type for contiguous sequences std::span does not provide a checked version of indexing and therefore should only be used via its iterator/range API. With the introduction of the at() function in C++ 26, a checked version of indexing is available. For more information on the use of std::span see [6.8](#HCB)

### 6.9.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects, C++ software developers can:

* Apply the avoidance mechanisms of ISO/IEC 24772-1:2022 clause 6.9.5.
* Follow the advice from clause [6.8.2](#HCB).
* Prefer the at() member function of the standard library containers over operator[] and handle the bounds violation exceptions.
* If indexing cannot be avoided, use static analysis or explicit checks to establish that bounds violations do not occur.

## 6.10 Unchecked Array Copying [XYW]

### 6.10.1 Applicability to language

The vulnerability as described in ISO/IEC 24772-1:2022 exists in C++, but can be mitigated using features provided by the language.

A buffer overflow occurs when some number of elements is copied from one buffer to another and the amount being copied is greater than is allocated for the destination buffer. This is a special case of 6.8 Buffer Boundary Violation [HCB]. The C library functions or hand-written loops for copying bytes or C-style strings are especially prone to this vulnerability.

As with clause 6.8 [HCB], in most cases the vulnerability can be avoided by using library classes, such as std::vector or std::string, which provide a copy operations operator that adjust the size of the target to fit the object being copied.

The standard library algorithms that copy into a target range can suffer from this vulnerability. In the case of potential overflow, the programmer must either ensure automatic extension of the underlying container, such as by using std::back\_inserter(container) as the output iterator, or ensure that the output range has sufficient space available. In the case of overlapping input and output ranges, the suitable copying algorithm must be selected, depending on the relative ordering of the ranges. In general, this situation can be avoided by using a more appropriate algorithm, for example, std::rotate.

For arrays with fixed sizes the assignment operator or copy-constructor of std::array is the means of safe array copying.

If a system requires its own container types with dynamic size, a naïve implementation might attempt to keep the copying external, like with C-style arrays. Such external copying should be avoided.

### 6.10.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects, C++ software developers can:

* Use the avoidance mechanisms of ISO/IEC 24772-1:2022 clause 6.10.5.
* Use standard library containers instead of C-style arrays, such as std::vector or std::array, that provide copying mechanisms that ensure the target array is large enough for the indicated source.
* When using standard library algorithms that copy into an output range, ensure that either the output range is extended automatically or has sufficient space available.
* Sidestep the need to use a copy algorithm on overlapping ranges, by selecting a more appropriate algorithm, such as std::rotate.
* If a system requires its own container types with dynamic size, ensure that their copy operations are correct and do not require loops or algorithms for copying their content.
* If copying C-style arrays is unavoidable, statically show that an access outside of the array cannot occur, or alternatively perform range checking before copying the array.
* Avoid using the copying functions from the headers

## 6.11 Pointer Type Conversions [HFC]

### 6.11.1 Applicability to language

The vulnerabilites as described in ISO/IEC TR 24772-1:2019 clause 6.11.1 applies to C++. In addition to pointers, C++ references are also vulnerable and the issues below include references when pointers are mentioned. In places where references cannot be substituted the corresponding code won’t compile.

In general casting pointers breaks the type system and should be avoided.

In C++, a C-style cast is defined in terms of the C++ cast operators const\_cast, static\_cast, and reinterpret\_cast. In addition, certain placement new expressions that do not initialize the storage can act like reinterpret\_cast. In some cases, it is unspecified which cast is used, for example, when a cast operation involves an incomplete type, a reinterpret\_cast may be used for the conversion which can produce an incorrect result.

Especially, reinterpret\_cast has the problem that it takes the original pointer value as a pointer of the target type rather than the original type. The C++ standard defines most cases where that happens as undefined behavior [[EWF]](#EWF). For example, the lifetime model of C++ might result in accessing the target type object outside of its lifetime. Other run-time issues can be caused by alignment violations. Using reinterpret\_cast<std::byte\*> to access the underlying memory of an object by casting its address permits access to the raw memory. However, casting the address of a piece of raw memory with the correct alignment and size to an object pointer and accessing that object is undefined behavior for most types, because doing so, will not start the lifetime of the object.

static\_cast only works, where conversion of the source type to the target type are related. However, with pointer types the compiler cannot always check that the actual object type corresponds to the desired target type, causing invalid casts. Naïvely assuming that addresses of a derived object and its base object are identical is wrong in most cases. For example, with multiple inheritance, the address of an object may be different than one of its base class sub-objects. Using the generic pointer type void\* (which is common in C APIs) allows converting between arbitrary pointer types using static\_cast. Most conversions via void \* where the originial object type and the final target type are different are undefined behavior in C++. C++ allows reinterpret\_cast to a pointer to an incomplete type or a static\_cast from void \* to a pointer to an incomplete type. Pointers to objects can implicitly convert to void \* (cv-qualified accordingly).

It is only defined to reinterpret\_cast the obtained pointer back to the original type. It is implementation-defined if that bidirectional void \* conversion also works for function pointers. A reinterpret\_cast can be used to convert a pointer from the integral types std::uintptr\_t/std::intptr\_t, but only if the value of the integer value was previously obtained by converting a valid pointer to said integral type. Casting an arbitrary integral value to a pointer is undefined behavior.

Casting along the inheritance relationships with dynamic\_cast is safe, but it requires the dynamic type to be known, which is the case when the types declare virtual member functions. Within a constructor or destructor only the static type of the current class is relevant, because the lifetime of any derived class object hasn’t started or has already ended. See subclause [Polymorphic Variables [BKK]](#BKK).

Conversions involving const and/or volatile properties of a type are permitted using const\_cast (see [Modifying constants [UJO]](#UJO)). Adding const with const\_cast is safe.

### 6.11.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects, C++ software developers can:

* Use the avoidance mechanisms of ISO/IEC 24772-1 clause 6.11.5.
* Avoid the C-style cast, reinterpret\_cast, and casts to and from void\*.
* For conversions that remove the const qualification, see the guidance in clause [Modifying constants [UJO]](#UJO).
* When downcasting or crosscasting see the guidance in [Polymorphic Variables [BKK]](#BKK).
* If pointer and reference conversions cannot be avoided, treat compiler warnings that are issued for such conversions as hard errors.
* Ensure through static analysis that arbitrary pointer casts return a type compatible with the source.

## 6.12 Pointer Arithmetic [RVG]

### 6.12.1 Applicability to language

The vulnerabilites described in ISO/IEC 24772-1:2022 clause 6.12.1 also apply to C++. The vulnerabilities caused by out-of-bounds access are covered in clause [6.8](#HCB).

Pointers to functions, pointers to members, and pointers to void do not allow pointer arithmetic.

The set of valid pointers referring to an array consists of the pointers to each array element plus the pointer just past the end of the array, however dereferencing a pointer one past the end of the array is undefined behaviour [[EWF]](#EWF). A pointer to a single object is considered to point to an array of size one with respect to pointer arithmetic.

Adding or subtracting an integral value to a pointer value must yield a result that is a valid pointer refering to the same array, otherwise the behavior is undefined. Note: the built-in indexing operator is defined in terms of pointer arithmetic.

Subtraction of two pointers has undefined behaviour unless both pointers refer to the same array or are both null.

Comparison of two pointers with one of the operators < > <= >= <=> has unspecified behaviour unless both pointers refer to the same underlying array or object. The standard library function objects for comparison, like std::less<>, provide a strict total order of pointers of a given type.

Iterators as defined by the standard library suffer from similar vulnerabilities as pointer arithmetic. Comparison and subtraction of two iterators, or computing std::distance cause undefined behavior for iterators that refer to different ranges. Forming an iterator that is outside of its underlying range similarly is undefined behavior.

### 6.12.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects, C++ software developers can:

* Avoid explicit pointer arithmetic, especially on a pointer to a single object.
* Prefer calling range-based algorithms over pointer arithmetic.
* Use the avoidance mechanisms of [6.8.2](#HCB).
* Compare arbitrary pointers to objects with the standard library comparison function objects instead of the operators < > <= >= <=>.

## 6.13 NULL Pointer Dereference [XYH]

### 6.13.1 Applicability to language

The vulnerability as described in ISO/IEC TR 24772-1:2019 clause 6.13 exists in C++. Dereferencing a pointer with the value of nullptr is undefined behavior [[EWF]](#EWF).

Using pointers is inherently problematic especially for function parameters and return values, due to the following issues:

* is a nullptr value valid in context?
* is there protection against dereference of a nullptr value?
* is it a single object or an array of objects? (T[] as a function parameter type is identical to T\*)
* what is the lifetime of the target of the pointer?
* who has ownership of the target object?

Using values instead of pointers sidesteps all pointer vulnerabilities, especially when returned from a function. For example, standard library containers like std::vector have value semantics and do not suffer from this vulnerability.

C++ references cannot be null in a well-defined program and solve the null-dereferencing vulnerability. They are particularly useful as function parameters. Using a reference as function return type requires the caller to avoid accessing an object outisde of its lifetime (see [[XYK]](#XYK), [[DCM]](#DCM)).

If absence of a value is necessary, a class type for optional values such as std::optional provides well-defined behaviour and single-object ownership. In case of attempting to access the value of an empty std::optional an exception is thrown.

Note: Be aware that optional<T&> is not supported by the standard library. For representing optional references std::optional<std::reference\_wrapper<T>> or a non-standard implementation of optional supporting references can be used.

If dynamically allocated objects are required, std::unique\_ptr<T> can be used for lifetime-management and for transferring ownership. When shared ownership of such objects is necessary, std::shared\_ptr<T> is a solution. Using std::shared\_ptr<T const> provides value semantics for immutable heap-allocated objects thus sidestepping most of the issues of pointers above. Constructing a smart pointer through the factories std::make\_unique or std::make\_shared will return a non-null smart pointer or throw an exception and thus prevent the vulnerability of null pointers, in contrast to legacy allocation mechanisms and some overloads of operator new. However, in general dereferencing a std::unique\_ptr or std::shared\_ptr equal to nullptr causes undefined behaviour, for example, when such a smart pointer is default constructed or a std::unique\_ptr is in a moved-from state. For further information see also [[XYL]](#XYL).

### 6.13.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects, C++ software developers can:

* Avoid the need for pointers by using values.
* Prefer references over pointers for parameters.
* Use std::optional instead of a pointer to represent a potentially empty value.
* Use containers with value semantics instead of plain arrays such as std::array, std::vector.
* Use make\_unique() or make\_shared() for dynamically allocating objects.
* If using objects of pointer-like types that may be a nullptr value is unavoidable, use the avoidance mechanisms of ISO/IEC 24772-1 clause 6.13.5.
* Use static and dynamic analysis tools to detect potential violations of this guideline.

## 6.14 Dangling Reference to Heap [XYK]

### 6.14.1 Applicability to language

The vulnerability as expressed in ISO/IEC TR 24772-1:2019 and ISO/IEC TR 24772-3:2020 C exists in C++. C++, however, provides mechanisms to mitigate the vulnerability. In contrast to C, where the mere existance of reachable memory for an object is sufficient to access it, the lifetime model of C++ makes it *undefined behaviour* [[EWF]](#EWF) to access an object outside of its lifetime. This results in undefined behavior, when an object access is attempted before one of its constructors is finished or after its destruction. For example, container types like std::vector or wrapper types like std::optional might have memory for an object available, that is not constructed or has ended its lifetime. For similar situations that result from accessing temporary objects or variables outside of their lifetime see subclause [[DCM]](#DCM). If such a temporary or local object manages heap memory (e.g., std::vector) referring to an element after the manager’s lifetime ended technically falls into the category of this vulnerability, but is covered there.

C++ provides a rich set of *pointer-like* types (potentially referring to heap memory) whose values may dangle, e.g.,

* References (the term value here includes references)
* Pointers
* Iterators
* Views, e.g.,
  + std::string\_view
  + std::span
* std::reference\_wrapper.

In addition, a user-defined class type can be a *pointer-like* type, if a subobject is of *pointer-like* type and refers to an object (*target*) whose lifetime is different from and not managed by the current object. Sometimes, regular object types act as *pointer-like* types, e.g., indices into a container or operating system handles, and their validity can not be directly mapped to the C++ object lifetime model.

If the lifetime of a *pointer-like* value ends before the lifetime of its *target*, then the vulnerability does not apply to that *pointer-like* value. This is the primary C++ strategy for avoiding vulnerabilities of dangling *pointer-like* values. For example, an object argument passed as a function parameter of reference type persists throughout the function call. The lifetime guarantee of a function argument passed indirectly via a *pointer-like* type does not apply if \* the target is destroyed explicitly by the called function (taking ownership of the target) or a concurrently executing operation, or if \* copies of the *pointer-like* parameter outlive the function call, for example, as the return value, or in a coroutine or thread frame.

For objects directly allocated on the heap C++ provides smart pointers and corresponding factory functions (e.g., std::make\_unique()) that allow transferring ownership or shared ownership to reduce the risk for dangling. However, storing the raw pointers managed by smart pointers can lead to accidental dangling, for example:

int \* f(){  
 auto up = std::make\_unique<int>(42);  
 return up.get(); // returned pointer dangles  
}

The C++ library containers, such as std::vector, manage the required heap memory for their elements. Referring to an element in a container via a *pointer-like* type is safe, as long as the container remains unchanged while the element object is accessed. In general, accessing an element in a mutated container via a *pointer-like* value obtained before the mutation is undefined behavior [[EWF]](#EWF). Different containers provide different validity guarantees of accessing an element via a *pointer-like* type that was obtained before a subsequent change in that container.

Hand-written loops are prone to attempt to access elements of a container that are non-existent, or have been relocated. Employing standard library algorithms to iterate over a range of elements from a container tends to be safer, as long as the underlying container is not accidentally changed. For example, the following code can cause a failure, due to the attempt to iterate over a changing std::vector:

std::vector v{1,2,3,4};  
copy(begin(v),end(v),back\_inserter(v)); // modifying v while iterating is undefined behaviour  
copy(begin(v),end(v), std::ostream\_iterator<int>(std::cout,", "));

A C++-specific way of causing dangling\_references to the heap is by means of a placement new expression (see 7.4 Placement new issues [XXX]).

### 6.14.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects, C++ software developers can:

* Use the avoidance mechanisms of ISO/IEC 24772-1 clause 6.14.5.
* Prefer value types that manage heap memory, for example, std::vector or std::string, and pass by value/return by value over the use of *pointer-like* types.
* If value types are not feasible, adopt a style that makes explicit the ownership and lifetime of heap resources, by using std::unique\_ptr or std::shared\_ptr or similar manager types and allocate heap memory exclusively with their corresponding factory functions std::make\_unique/std::make\_shared.
* Ensure all copies of any *pointer-like* value are no longer accessible or accessed after the referent’s lifetime ends.
* Prohibit access of a container through an invalidated iterator, and similarly, access via other potentially dangling *pointer-like* values, e.g., views.
* Avoid unmanaged heap allocation by new/delete expressions or calling C-library heap allocation functions.
* Avoid manually ending the lifetime of an non-owned object by explicitly calling its destructor or std::destroy\_at.
* Use static and dynamic analysis tools to detect dangling.

## 6.15 Arithmetic Wrap-around Error [FIF]

### 6.15.1 Applicability to language

C++ has the vulnerability as documented in ISO/IEC 24772-1 clause 6.15, since overflow situations are undefined behaviour [[EWF]](#EWF) for signed integer arithemtic and wrap-around for unsigned integer arithmetic, which can lead to surprising results. C++ specifies that

* although signed integers are defined to be represented as two’s complement, signed integer overflow is undefined behaviour,
* unsigned integer arithmetic is defined to wrap-around (modulo arithmetic) and not overflow,
* expressions of integral type can implicitly convert to other integral types, even when passed as function arguments,
* operands of different arithmetic types in an expression are subject to the implicit “usual arithmetic conversions” that can be confusing, and
* integral promotion can change the type of an expression from an unsigned type to a signed type before arithmetic takes place.

For example, integral promotion happens when multiplying two unsigned short operands which can result in undefined behavior:

auto f(){  
 std::uint16\_t x{50'000},y{50'000};  
 return x \* y; // undefined behaviour due to overflow, returns int  
}

In the above, for a 16-bit short and a 32-bit int, i.e., std::numeric\_limits<int>::max()==0x7fff'ffff, x and y are promoted to int and the multiplication then overflows which is undefined behaviour.

Even when operands have the same unsigned type, wrap-around arithmetic can be confusing, for example, 4U - 5U yields a large positive value.

Calling a function taking a parameter of integral type with an argument of different integral type works due to implicit conversions. If a different overload with a better match becomes visible the called function can change when re-compiled (see [6.21 Namespace Issues[BJL]](#BJL))

Using brace-initialization prevents implicit narrowing conversions in contrast to other forms of initialization. For example:

std::uint16\_t x{500'000}; // won't compile due to narrowing  
std::uint16\_t y = 500'000; // compiles, but truncates value

The mitigations for wrap-around errors in C++ are different than for C. The type system of C++ allows user-defined class and enum types with corresponding overloaded operators. Such user-defined types can individually control which implicit conversions or mixed type arithmetic they support, if any. For example, one can force arithmetic to be done with unsigned types:

enum class uint16: std::uint16\_t{};  
uint16 operator\*(uint16 a, uint16 b){  
 return static\_cast<uint16>(static\_cast<unsigned>(a) \* static\_cast<unsigned>(b));  
} // guarantee wrap-around

High-integrity software using the built-in integral types should

* use integral types with a well-known range (i.e., those from <cstdint>),
* refrain from relying on integral promotion and implicit conversions, and
* guarantee that allowed variation in the range of representable values do not influence computed results.

### 6.15.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects, C++ software developers can:

* Use the avoidance mechanisms of ISO/IEC 24772-1 clause 6.15.5.
* Use appropriate user-defined types with well-defined range in place of built-in integral types, that mitigate against implicit conversions and undefined behaviour.
* Use types with a well-defined range and have compile-time guarantees that the range of types used is sufficient, i.e., with static\_assert.
* Avoid implicit integral promotions and integral conversions, especially on function arguments.
* Avoid mixing integral types with different size or different signedness in the same expression.
* Ensure that the result of any mathematical operation fits within the constraints of the types involved within the expression.
* Use unsigned types that do not promote to int to avoid undefined behavior due to signed integer overflow.
* Consider the use of numeric\_limits<T>::is\_modulo to determine whether or not an integer type T wraps for the target system.
* Document where wrap-around is expected for a type.
* Use static and dynamic analysis tools to detect problematic expressions.

## 6.16 Using Shift Operations for Multiplication and Division [PIK]

### 6.16.1 Applicability to language

The vulnerability as described in ISO/IEC 24772-1 clause 6.16 exists in C++. C++ complicates the discussion in 24772-1 clause 6.16 as a result of the integral promotion (see clause 6.06 [FLC]). A left-shift on an operand that gets promoted can result in a value outside the operand’s unpromoted type’s range.

Not every use of a shift operator is a bit-shift due to operator overloading.

### 6.16.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects, C++ software developers can:

* use the avoidance mechanisms of ISO/IEC TR 24772-1:2019 clause 6.16 \*Using Shift Operations for Multiplication and Division [PIK].
* For promoted operands, consider converting the result of a shift operation back to the original type of the left hand operand (say using a static cast or an assignment to an object of that type) before using the result in further operations.

## 6.17 Choice of Clear Names [NAI]

### 6.17.1 Applicability to language

The vulnerability as described in ISO/IEC TR 24772-1:2019 clause 6.17 is applicable to C++, as it is susceptible to errors resulting from the use of similarly appearing names. However, the language rules prevent using an identifier that has not been declared. There are two possible issues: the use of the identical name for different purposes (see clause [6.20 Identifier Name Reuse [YOW]](#YOW)) and the use of similar names for different purposes.

* *Mixed casing*: C++ distinguishes upper case and lower-case letters in names. Thus, confusion can arise through the use of Item and item as distinct declared identifiers with different meanings.
* *Underscores*: C++ permits single underscores in identifiers and they are significant. Thus, BigDog and Big\_Dog are different identifiers. Multiple underscores or a leading underscore followed by an upper-case letter cause undefined behaviour [[EWF]](#EWF), thus Big\_\_Dog and \_BigDog should not be used.
* *Singular/plural forms*: C++ does permit the use of identifiers which differ solely in this manner such as item and items. However, C++ lets the programmer use the identifier item for a single object of a type T and the identifier items for an object denoting a range of items of T. The use of item where items was intended or vice versa usually will be detected by the compiler because of the type mismatch.
* *International character sets*: C++ compilers support unicode characters in their source character set and also in identifiers. This can lead to vulnerabilities due to (near) identical representation of distinct names (e.g., homoglyph attacks). Another potential problem is in changing the text direction from left-to-right to right-to-left. This can render code that the compiler accepts, unintelligible, and can be used as an source-code attack vector.
* Additionally, in Unicode, the right-to-left character is encoded at U+200F RIGHT-TO-LEFT MARK (‏). In UTF-8 it is E2 80 8F. Usage is prescribed in the Unicode Bidi (bidirectional) Algorithm.[1]
* *Identifier length*: The C++ standard recommends compilers to distinguish at least 1024 significant characters for an identifier. The actual limit of compiler for identifier length is implementation-defined and thus must be documented. All characters up to the length supported by the compiler in an identifier in C++ are significant. Thus Long\_IdentifierA and Long\_IdentifierB are always different if the documented identifier limit is at least 16 characters.

C++ permits the use of names such as x, xx, and xxx, possibly defined in non-obvious scopes, and a programmer can easily, by mistake, write xx where x or xxx was intended. Especially for overloaded functions, argument-dependent-lookup might find a function in a scope that the user did not consider. The use of the wrong name will typically result in a failure to compile so no vulnerability will arise. However, if the wrong name has a type compatible with the intended name’s type, then an incorrect executable program will be generated.

C++ defines reserves some names as context-specific keywords. While it is technically possible to use those names for other purposes, such use can be confusing.

In the global scope some namespaces (such as std, posix) are reserved and should not be used otherwise.

### 6.17.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects, C++ software developers can:

* Use the avoidance mechanisms of ISO/IEC 24772-1:2019 clause 6.17.5.
* Follow the rules of ISO/IEC 14882:2020 clause [lex.name] regarding names to refrain from usage.
* Avoid the use of similar names to denote different objects of the same type.
* Use consistency in choosing names, especially for dealing with similar names.
* Use static analysis and tooling to enforce project-specific naming rules and detect possible similar names, for example, homoglyphs and unexpected text-direction vulnerabilities.
* Keep the scope of names as small as reasonable.
* Ensure that the names in your program do not exceed the compiler’s documented limit.
* Do not differentiate names through only a mixture of case or the presence/absence of an underscore character.
* Do not use as identifiers the contextual keywords final , import, module and override and other reserved names.

## 6.18 Dead Store [WXQ]

### 6.18.1 Applicability to language

The vulnerability as documented in ISO/IEC TR 24772-1:2019 clause 6.18 exists in C++.

The language definition permits the compiler to eliminate effects of the abstract machine that are not observable, in particular, dead stores. For example, the often-attempted write operations to non- volatile member variables in a destructor that are not subsequently used can be elided by the compiler.

C++ compilers and static analysis tools do exist that detect and generate warnings for dead stores.

The error in ISO/IEC 24772-1:2019 subclause 6.18.3 that the planned reader misspells the name of the store is possible but unlikely in C++ since the language specifies that all objects shall be declared and typed, and the existence of two objects with almost identical names and compatible types (for assignment) in the same scope would be readily detectable. See 6.17 [NAI] Choice of clear names, 6.20 Identifier name reuse [YOW], and 6.21 Namespace issues [BJL]

### 6.18.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects, C++ software developers can:

* Use the avoidance mechanisms of ISO/IEC 24772-1 clause 6.18.5.
* Use compilers and static analysis tools to identify dead stores in the program.
* Provide sufficient synchronization to non-const variables that are accessed by multiple execution agents. See clause 6.61 Concurrent data access [CGX]

## 6.19 Unused Variable [YZS]

### 6.19.1 Applicability to language

The vulnerability as documented in ISO/IEC 24772-1 clause 6.19 exists in C++.

A common practice for resource management in C++ relies on what is called “RAII” or “SBRM” (scope-based resource management): employing a class’ destructor to release resources managed by the object. This can lead to code without visible use of a variable being present in the source code, because all work is done by the variable’s constructor and/or destructor.

### 6.19.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects, C++ software developers can:

* Use the avoidance mechanisms of ISO/IEC 24772-1 clause 6.19.5.
* Use compiler warnings and/or static analysis tools to indicate and eliminate unused variables.

## 6.20 Identifier Name Reuse [YOW]

### 6.20.1 Applicability to language

The vulnerability as described in ISO/IEC TR 24772-1:2019 clause 6.20 exists in C++, except for the second issue of limited identifier length. In C++ all characters in an identifier are significant.

C++ provides the scope resolution operator ::{.cpp} to access identifiers from non-local scopes.

Overloading and specialization of functions is a cornerstone of C++ generic programming. In this context, the reuse of function names is essential. See clause 6.41 for inheritance issues associated with name reuse.

Overloaded function names and operators considered in an expression are not restricted to a simple scope hierarchy, because of argument-dependent lookup (ADL). In generic code the unqualified function or operator selected can come from a scope based on the type of the arguments and not from the current scope hierarchy. The rules for which namespaces are eligible for lookup of unqualified functions and operators are intricate, but required to make overloaded operators work.

In addition, if implicit conversions can happen on arguments, the overload selected by ADL can be different from programmer expectation even in non-generic code, especially when an argument is of a type that can be implicitly converted to another type where a corresponding overload is defined. Visibility on a namespace-level of such an operator overload may make it eligible, even if neither argument matches the parameter types directly. In the best case this leads to a compile error due to ambiguities, but it can also result in perfectly compiling code executing an unexcepted overload.

The following example demonstrates part of the problem:

#include <iostream>  
#include <typeinfo>  
  
namespace Y {  
template <typename T>  
void print(T i){  
 std::cout << typeid(T).name()<< ":" << i ;  
}  
template <typename T>  
void println(T x){  
 print(x); // expects to call Y::print  
 std::cout<<'\n';  
}  
}   
namespace X {  
 struct A{  
 A(double){}  
 friend // make this a hidden friend  
 std::ostream & operator << (std::ostream & out, A const &a){  
 return out << "An A as expected\n";  
 }   
 };  
 void print(A a){ // not expected to be called by println  
 std::cout << "Surprise happens!";  
 }  
}  
int main(){  
 X::A a{3.14};  
 Y::println(42); // i:42 - calls Y::print  
 std::cout << a; // An A as expected - calls X::operator<<  
 Y::println(a); // Surprise happens! - calls X::print  
 Y::println(42u);// u:42 - calls Y::print  
}

The above code calls the overload print(A) from println since it is pulled in by ADL. On the other hand, ADL is required to work to allow the output operator for type X::A to work.

The consideration of implicit conversions together with ADL can be suppressed by defining operator overloads as class members or as *hidden friends*. The latter is achieved by declaring all corresponding overloads as friend functions in the class that take the class’ objects as arguments. Generic base classes can provide mix-in facilities for *hidden friends* by taking the argument type that is the derived class as template parameter.

### 6.20.2 Avoidance mechanismsfor language users

To avoid the vulnerability or mitigate its ill effects, C++ software developers can:

* Use the avoidance mechanisms of ISO/IEC 24772-1 clause 6.20.5, with the exclusion of guidance related to truncated identifiers.
* Qualify names to disambiguate potential conflicts between names introduced from different scopes.
* Document argument-dependent lookup usage where name qualification is not desirable.
* Limit the visibility of overloaded operators or functions for class types by defining them as member functions or *hidden friends*.
* Place overloaded operators that are not class members and cannot be provided as *hidden friends* together with their argument type in a namespace that is not the global namespace,so that they are picked up by ADL.
* Use modern integrated development environments that inform about the declaration of any identifier occurrence.
* Enable compiler diagnostics that inform about the hiding of declarations.
  + DCL60-CPP. Obey the one-definition
  + rule (6.21)
  + DCL40-C. Do not create incompatible declarations of the same
  + function or object (6.21)

## 6.21 Namespace Issues [BJL]

### 6.21.1 Applicability to language

The vulnerability described in ISO/IEC TR 24772-1:2019 clause 6.21 exists in C++. It can occur in particular when a used library changes its API. The situations where it exists are related to the following cases:

* Template specialization, where clause 6.40 [#SYM] applies;
* Overloading, where clause 6.20 [#YOW] applies;
* Overriding, where clause 6.41 [#RIP] applies.

In the case of template specialization or non-identical definitions of the same entity in different translation units (ODR-violation), ill-formed code might be the result, however, a C++ compiler is not obliged to diagnose that situation, leading to undefined behaviour [[EWF]](#EWF).

In the case of overloading and overriding cases, C++ compilers are required to diagnose an ambiguity if it exists.

However, overload resolution applies preference rules in order to select among multiple matching functions or function templates as a means to resolve the ambiguity among these functions. Hence, for calls that are not perfect matches, the user cannot guarantee in the presence of later changes which function is called, as another, better match can be introduced subsequently. The call in question then changes its binding without warning upon its next compilation. For cases, where the preference rules do not resolve the ambiguity, the resulting error message by the compiler avoids the vulnerability. Function template specializations are not considered during overload resolution, only the base template is considered.

void foo (long);  
  
// void foo (int);  
   
void bar ()  
{  
 foo (0); // The call to 'foo(long)' requires implicitly conversion  
 // from 'int' to 'long'. The function 'foo(int)'   
 // would be a "better match" and so would silently  
 // be chosen when subsequently introduced  
}

A new declaration can impact existing code in a number of situations involving the addition of:

* an overload,
* a using directive,
* a declaration into a namespace that is searched as part of “Argument-Dependent Lookup”,
* a non-template function that is preferred to a function template,
* a template specialization.

A using directive broadens the possible scopes that will be examined for names during lookup. Where lookup searches a namespace referred to by a using directive, all names in that namespace will be visible some of which may be unwanted. A using declaration, on the other hand, declares only the specified name into the scope of the using declaration.

namespace NS1  
{  
 void f1 (int);  
 void f2 (int); // Added later  
}  
namespace NS2  
{  
 using namespace NS1; // 'f1' needed  
 void f2 (long);  
 void bar ()  
 {  
 f1(0); // Calls 'NS1::f1'  
 f2(0); // Unintentionally calls 'NS1::f2'  
 }  
}  
namespace NS3  
{  
 using NS1::f1; // 'f1' needed  
 void f2 (long);  
 void bar ()  
 {  
 f1(0); // Calls 'NS1::f1'  
 f2(0); // Calls 'NS3::f2' as expected  
 }  
}

Overload resolution only considers conversions for the explicitly specified arguments and does not take default parameters into account:

void f1 (short, int = 0);  
void f1 (int, short = 0);  
  
void f2 ()  
{  
 f1 (1); // calls 'f1(1, 0)' as '1 -> int' is better match than '1 -> short'  
 f1 (1, 0); // ambiguous, ill-formed, won't compile  
}

The following example demonstrates a situation where the late addition of a better matching overload causes a silent change in the semantics of an existing program.

namespace NS  
{  
 struct A  
 {  
 };  
  
 template < typename T > T foo ( T t )  
 {  
 return t;  
 }  
}  
  
namespace NS2 // separately developed and included from a header file  
{  
 struct B  
 {  
 };  
// This code will be added later  
// template < typename T > T \* foo ( T \* t )  
// {  
// return t;  
// }  
}  
  
using namespace NS2;  
using namespace NS;  
  
void bar()  
{  
 A \* a;  
 B \* b;  
 foo (a); // After the commented-out code is added to NS2, the binding of foo changes silently from NS::foo to NS2::foo  
}

This issue can be avoided by avoiding using namespace xxx{.cpp} and explicitly qualifying each call, such as NS::foo(a){.cpp}.

Analogously, when a more specialized template is added to an imported namespace where the more general template has already been provided in another namespace, preference rules will silently prefer the more specialized template.

A similar situation can occur when a conflict arises between compiler-synthesized or rewritten operators and explicitly created versions of those operators, as in the following example.

struct A  
{  
 bool operator==(A const &) const { return true; }   
};  
  
// Evil hijacking of !=  
// bool operator != (A const &, A const &) { return true; } // #1  
  
void bar (A const & a) {  
 a != a; // #2  
}

In the above example, the declaration of operator== will have a corresponding synthesised operator!= generated by the compiler, since there is no suitable user-declared !=. If the operator!= becomes visible, then the code at #2 uses the user-declared operator!= instead of the synthesized one, which can lead to a silent and unexpected change of behaviour. This is particularly risky when the operator is declared outside of the immediate visibility of the original definition.

### 6.21.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects, C++ software developers can:

* Use the avoidance mechanisms of clauses 6.20.2, 6.40.2, and 6.41.2 as applicable.
* Consider using fully qualified names for calls that rely on an implicit conversions.
* Arguments to called functions should not be subject to implicit conversions.
* Prefer using declarations to using directives.
* Do not overload and use default arguments for the same set of functions.
* Do not specialize function templates.
* For template specialization, ensure that specializations are declared as follows:
  1. In the same file as the primary template; or
  2. In the same file as the user-defined type for which the specialization is declared.
* Define an entity in only one file to prevent ODR-violations.
* Ensure that no ODR-violations occur, i.e., through a static analysis tool.
* Use a version-aware analysis tool to identify situations where preference rules cause a silent change of name binding between versions.
* Only declare equaility or relational operators as member functions or friend functions of the class

## 6.22 Missing Initialization of Variables [LAV]

### 6.22.1 Applicability to language

The vulnerability as described in ISO/IEC TR 24772-1:2019 exists in C++, however, defining variables only when they can be initialized properly with an initializer in their definition avoids reading uninitialized memory.

Defining/allocating objects of trivial type with automatic/dynamic storage duration without initialization leaves the object with an indeterminate value. A subsequent read of such a variable before it has been written is undefined behavior [[EWF]](#EWF). In addition, sub-objects of trivial type that are omitted in a constructor’s member initializer list and not initialized by the constructor’s body or by a *default member initializer* will not be initialized by that constructor. For example, the following class definitions suffer from incomplete initialization of subobjects, even though class test defines a default constructor:

struct base { short num; };   
struct test : base   
{   
 enum E1 { e1a=100, e1b, e1c };   
 int one;   
 int two;   
 double ar1[2]{ 1.1, 2.2 };   
 double ar2[2];   
 E1 e1;   
 E1 e2;   
 test()   
 : // base unintialized   
 one{ 1 }  
 // two uninitialized  
 // ar1 initialized through default member initializer  
 // ar2 uninitialized  
 , e1{} // initializes to zero, not a named enumerator value   
 // e2 uninitialized  
 { }   
};

Dynamically allocating memory for an object using malloc, or some other C-style equivalent, does not initialize the object. Interpreting such memory as an object with trivial type will result in it having an indeterminate value.  
Objects with non-trivial type require running a constructor for its lifetime to start correctly. In both cases attempting to cast a pointer to the allocated memory and using the object is undefined behavior except for special sanctioned cases, see [Conversion Errors [FLC]](#FLC) and [Pointer Type Conversion [HFC]](#HFC).

Non-local variables with static storage duration that are dynamically initialized can cause undefined behavior if the initialization depends on other such variables. If the dependency is in the same translation unit the sequencing is defined in definition order, however, there is no sequencing guarantees across translation unit boundaries and thus, undefined behaviour can occur by accessing an uninitialized variable. For example:

struct A {  
 A (int i ) : i\_ { i } { }  
 int i\_;  
};  
struct B {  
 B (A const & a) : j\_{a.i\_} { }  
 int j\_;  
};  
extern A a; // declare existance of variable 'a'  
// defining variables with dynamic intialization:  
B b { a }; // #1  
A a { 42 }; // #2

If #1 and #2 are in the same translation unit, then a in #1 is incompletely initialized (zero initialized). If #1 and #2 are in different translation units, then the order of initialization of a(#2) relative to b(#1) is indeterminate.

The constexpr-specifier for a variable ensures initialization at compile time. The constinit specifier ensures a variable is initialized at compile time, even if it is non-const.

Defining non-member variables as const or as constexpr, enforces initialization by the compiler and makes reasoning about code easier.

If determining the initial value of a variable requires complex logic, putting that logic into an immediately-invoked lambda expression that computes the initial value, permits the variable to be initialized when defined.

See C++ Core Guidelines ES.20 and CERT C++ Coding Guidelines EXP53-CPP. Note that ES.20 and EXP53 are complementary. Both point out that you should always initialize before reading, but ES.20 uses the narrow sense of initialize while EXP53 includes assignment.

### 6.22.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects, C++ software developers can:

* Use the avoidance mechanisms of ISO/IEC 24772-1 subclause 6.22.5
* Initialize every object defined, including data members, such as by providing an initializer for each variable definition.
* Consider defining non-member variables as const to enforce initialization, otherwise defer the definition of the variable until a sensible initial value can be provided.
* In situations where variable initialization seems to require complex logic, encapsulate that logic either in a function or a lambda that encapsulates the complex logic and returns the initial value. In case of multiple interdependent variables, group those in a struct and return the struct.
* Avoid using non-const variables with static storage duration (see 6.61 Concurrent Data Access [CGX]).
* Define variables with static storage duration with constexpr or constinit whenever possible to avoid problems with dynamic initialization order.
* Do not depend on the order of dynamic initialization of non-local variables with static storage duration, unless the initialization order is guaranteed by the language.
* If the use of uninitialized variables is unavoidable, employ tools, including the compiler, that detect reads of variables with indeterminate values.

## 6.23 Operator Precedence and Associativity [JCW]

### Applicability to language

The vulnerability as described in ISO/IEC TR 24772-1:2019 clause 6.23 is applicable to C++.

Operator precedence and associativity in C++ are determined by the C++ grammar. There are four operators that cannot be overloaded (user-defined):

* :: ;
* . ;
* .\* ; and
* ?: .

Due to the large number of operators, one is recommended to consult an operator precedence table when needed, e.g., [[https://en.cppreference.com/w/cpp/language/operator\_precedence]](https://en.cppreference.com/w/cpp/language/operator_precedence). For example, in C++, the bitwise logical and shift operators are sometimes incorrectly treated as having the same precedence as arithmetic operations even though the bitwise operators have lower precedence. For example, the following (correct) expression subtracts one from x and then checks if the result is zero:

x - 1 == 0

which is equivalent to (x - 1) == 0,i.e., x - 1 is done first, then that result is compared to 0. Programmers mistakenly thinking the bitwise operations have the same precedence as arithmetic ones might write:

x & 1 == 0

intending to perform (x & 1) == 0, but precedence rules result in this evaluating x & (1 == 0) instead. This would have been easily fixed by using parenthesis to ensure the proper evaluation of an expression.

In addition to the aforementioned, C++ also permits operators to be overloaded when used with user-defined types. While it is not possible to change the precedence, associativity, and number of operands of overloaded operators [C++17, Clause 16.5 [over.oper], para. 6], overloaded operators can be executed differently than built-in operators. For example, overloaded operators lose any built-in operator short-circuiting properties and sequence order guarantees. Similarly overloaded operators and their arguments' evaluations behave as normal function calls, differing from built-in operator evaluation.

struct A { };  
bool operator&&(A const &, int);  
int foo ();  
  
void bar (A const & a)  
{  
 if (a && foo()); // 'foo()' always evaluated  
 if (false && foo()); // 'foo()' never evaluated  
 if (a.operator&& (false,foo())); // 'foo()' always evaluated  
}

Note that overloaded assignment falls into this category.

For issues related to the declaration of equality and relational operators see Clause 6.25 [KOA].

### 6.23.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects, C++ software developers can:

* Use the avoidance mechanisms of ISO/IEC 24772-1 Clause 6.23.5 [JCW].
* Enable all C++ compiler/tool warnings and static analysis tool diagnostics concerning possible issues with precedence and associativity to help detect and avoid mistakes.
* Even if technically unnecessary, use parentheses around operator (sub)expressions that are known to be or felt likely to be sources of error.
* Avoid overloading the following operators &&, ||, , and their alternative spellings.
* Break up complex expressions and use temporary variables to make complex expressions easier to understand and maintain.

## 6.24 Side-effects and Order of Evaluation of Operands [SAM]

### 6.24.1 Applicability to language

The vulnerability as described in ISO/IEC TR 24772-1:2019 clause 6.24 exists in C++.

The evaluation of an expression includes: (i) its value computation; and (ii) its side-effects. The value computation is the value returned by the expression, e.g., the valuation of 3 \* 2 + 1 is 7. The side-effect of an expression are

* Write accesses to objects in that expression,
* Reading a volatile object,
* Calling a library I/O function, and
* Calling a function that does any of the above.

For example consider:

int i = 2;  
int j = i++;

the evaluation of i++ is 2 and the side-effects are the writing of 3 to i and the initialization of j.

Within an expression, one must ensure an object is stored only once to avoid undefined behaviour [[EWF]](#EWF), e.g.,

i = i++ + 5; // undefined behaviour (before C++17)

or

k = i++ + i\--; // undefined behaviour in all versions of C++

and expressions modifying objects can only read the object to determine the value to be stored (e.g., ++i requires reading the value), i.e., other accesses are undefined behaviour, e.g.,

my\_array\[i\] = i++; // undefined behaviour (before C++17)  
  
Starting with C++17, the evaluation order of an expression involving  
overloaded operators preserves the sequenced before behaviour of the  
built-in operator:  
  
```{.cpp}  
my\_array[i] = i++;  
my\_array[i++] = i++;

say i = 10 before the expression

evaluate RHS i++ – i is 11

evaluate my\_array[i++] //evaluates my\_array\[11\], then assigns i to 12

my\_array[11] is assigned 10

This occurs because assignment is sequenced after the value computation of the right and left operands and before the value computation of the assignment expression and, the right operand is sequenced before the left operand. [C++17, Clause 8.18 [expr.ass], para. 1] Since this is the built-in operator, this statement can be thought of as:

Compute value of right-hand-side: i++ (e.g., integer value).

Compute value of left-hand-side: my\_array[i] (e.g., memory address).

Apply side-effects of i++.

Apply side-effects of the assignment.

In general, one should follow commonly-stated C/C++ advice of never reading from and writing to the same object within an expression to avoid potential vulnerabilities. Often breaking the expression into separate statements achieves clear and clean semantics, e.g.,

++i;  
my\_array[i] = i;

or

my\_array[i] = i;  
++i;

makes it unambiguous what the value of i is during the array assignment and eliminates the possibility of vulnerabilities.

In addition, it is important to note that overloading an operator disables short-circuiting behaviours (e.g., built-in boolean operators): those operators' operands are all evaluated before the operator itself.

The C++ built-in (two-argument) Boolean operators (e.g., && and \|\|)as well as <type\_traits>’s std::conjunction and std::disjunction operations are all short-circuiting, i.e., if the value of an earlier (from left-to-right) operand of an operation determines the result of the operation, then all remaining arguments are not evaluated.

<!--  
Conjunction and disjunction operate at compile time and the short-circuiting is about template   
instantiations that might lead to compile errors otherwiese. This is not a runtime safety issue. I   
suggest dropping that (Peter)\_  
-->

Typically this allows one to write code like this, e.g.,

int \*p;  
 // ...  
 if (p != nullptr && \*p != 0) {  
 /\* do something \*/  
 }

i.e., if p is nullptr, then \*p != 0 is never executed, thus, avoiding undefined behaviour. Only when p is not nullptr is \*p != 0 is evaluated. It must be stressed that this only applies to the built-in && and || operators: user-defined operator overloads as functions always evaluate all operands first.

Consequently should one want to always evaluate all operands of a boolean expression, one should not write code like this:

bool x = foo() && bar();

where foo() and bar() are functions that return something convertible to bool. In this expression, if foo() returns false, then bar() will never be executed; –only when foo() returns true will bar() be executed. Similarly for ||:

bool y = foo() || bar();

i.e., only when foo() returns false will bar() be executed if foo() returns true then bar() will never be executed. Thus, if both foo() and bar() are both required to be executed, then execute them in separate statements first, e.g.,

bool foo\_result = foo();  
bool bar\_result = bar();  
bool x = foo\_result && bar\_result;  
bool y = foo\_result \|\| bar\_result;

<!--  
Stephen: My write-up here is lengthy but should help get more terse  
wording\... but I note this: C++ operator information is in C++17 Clause  
8 and Clause 16.5, \... Also per 16.5.1 para 2. unary and binary forms  
of the same operator are considered to have the same name so one can  
hide another from an enclosing scope. Thus, this is also another  
possible vulnerability.\]  
-->

### 6.24.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects, C++ software developers can:

* Use the avoidance mechanisms of ISO/IEC 24772-1 clause 6.24.5.
* Write expressions so that the same effects will occur under any order of evaluation that C++ permits since side effects can be dependent on an implementation specific order of evaluation.
* Avoid overloading logical operators (&& and ||), as these overloaded versions will not short-circuit.
* *Ensure that* ++, --, @= *are used as solitary items in a statement.*
* Be aware to which C++ standard a compiler is compiling code against. This determines the semantics of all operator (overloaded and built-in) expressions.
* Avoid placing multiple operations with side effects in a single expression.
* Write simple code expressions and statements so that within any expression or statement an object is one of:
  + Read from;
  + Only modified once; or
  + Only read and modified by: a single increment; a single
  + decrement; a single simple assignment; or a single compound assignment.
* Be aware that different versions of C++ have different evaluation order specifications and program accordingly.

## 6.25 Likely Incorrect Expression [KOA]

### 6.25.1 Applicability to language

The vulnerability as described in ISO/IEC TR 24772-1:2019 clause 6.25 exists in C++.

C++ has several instances of operators which are similar in structure, but different in meaning. Examples of operators in C-based languages that can cause confusion are:

* == and =;
* && and &;
* || and |;
* <, <<, and <<=; and
* >, >>, and >>=.

The typographical similarity can lead to code like the following, where it is unclear if the expression as spelled is actually intended, or if the author has typos in it, meaning a different operator instead:

auto f(unsigned i, unsigned j)  
{  
 return (i > 1) & (j = 1); // (>>, &&, ==)?  
}

The following code in a production phone OS caused the “bricking” of many users phones:

if (key\_data\_.has\_value() & !key\_data\_->label().empty())

instead of

if (key\_data\_.has\_value() && !key\_data\_->label().empty())

or the even clearer using the alternative operator representation and for &&

if (key\_data\_.has\_value() and !key\_data\_->label().empty())

As a general rule, the use of =, +=, -= in an expression when the operator is not the final assignment to a variable is unsafe since the assignment operator creates side-effects within the expression which are difficult to analyze by a human reader and can be have different results depending upon the order of evaluation of terms within the expression.

But even in assignment expression flipping the assignment symbol with the operator can itself lead to valid code that was not intended:

int i{42};  
  
i += 22; // i becomes 64  
i =+ 22; // i becomes 22  
i =- 22; // i becomes -22

C++ provides significant freedom in constructing statements. This freedom, if misused, can result in unexpected results and potential vulnerabilities.

Since the order of evaluation within expressions is only partially defined, sub-expressions with side effects on variables used within the overall expression can result in undefined behaviour [[EWF]](#EWF).

The flexibility of C++ can obscure the intent of a programmer. Consider:

int x,y;  
/\* ... \*/  
if (x = y){  
 /\* ... \*/  
}

A fair amount of analysis may need to be done to determine whether the programmer intended to do an assignment as part of the if statement (valid in C++) or whether the programmer made the common mistake of using an = (assignment) instead of a == (equality).

This confusion can be corrected by moving assignments outside of Boolean contexts. This would change the example code to:

int x,y;  
/\* … \*/  
x = y;  
 if (x == 0) {  
 /\* ... \*/  
 }

This would clearly state what the programmer meant and that the assignment of y to x was intended.

Additional confusion occurs in the use of the logical && or || operators and the bitwise & or | operators. The compiler will implicitly convert arithmetic expressions to bool for operands of the logical operators. Similarly, operands of bool type will be promoted to int for operands of the bitwise operators (see [Conversion Errors [FLC]](#FLC)).  
It may not be clear whether the programmer intended to use the logical operator && or bitwise operator & instead:

unsigned f(unsigned i, unsigned j)  
{  
 return (i > 0) & j;  
}

Using the alternative tokens and / or in lieu of && and || reduces the possibility of confusion. Similarly, a not\_eq b is preferable to a != b since the latter is easily confused with the equally valid expression a |= b.

Programmers can easily get in the habit of inserting the ; statement terminator at the end of statements. However, inadvertently doing this can drastically alter the meaning of code, even though the code is valid as in the following example:

int a,b;  
 /\* … \*/  
 if (a == b); // the semi-colon will make the following code always execute  
 {   
 /\* ... \*/  
 }

Because of the misplaced semi-colon, the code block following the if will always be executed. In this case, it is extremely likely that the programmer did not intend to put the semi-colon there.

Unary ‘+’{.cpp} on a variable is (almost) a no-op, and is possibly a mistype of ‘++’{.cpp}. A unary ‘-’{.cpp} on a variable will switch its sign, unless applied to a variable of an unsigned type, in which case the result is the value subtracted from 2^n where n is the number of bits in the unsigned type.

#### C++ overloading of operators can also cause confusion.

The language does not impose any restrictions on semantics of overloaded operators. This can cause (potentially generic) code to behave in completely unobvious ways, when such types with “unusual” operator semantics are used.

For example, the boost.spirit library allows code like the following to create parser rules:

r = real\_p >> \*(ch\_p(',') >> real\_p); // rule that accepts a comma-separated list of real numbers

This library uses C++ operator overloads to create an embedded domain-specific language for grammar rules, allowing the specification of parser rules as C++ expressions.

When overloaded, related operators like the compound assignment with their base operator are not longer guaranteed to keep their behavioral relationship that they have for built-in types. For example, a += b is not guaranteed to behave like a = a + b, or being defined at all.

Similarly for overloaded relational operators, for a == b, there is no guarantee that a != b is equivalent to !(a == b) if both are overloaded by the user.

Unless all relational operators for a type are defined either explicitly in a consistent way or implicitly, unexpected results can occur. A user-declared three-way comparison operator (<=>) is used by the compiler to synthesize the relational operators consistently. If operator<=> is defined as =default, the equality comparison operators will also be defined; and if operator== with return type bool is defined, a corresponding inequality operator!= is also defined implicitly.

#### 6.25.1.2 The Most vexing parse

In C++ if the syntax of a declaration can be interpreted as a function declaration in preference to an object declaration, . In the following example, the intention is to create a scoped lock object, but the compiler interprets it as a function declaration lock(). ```{.cpp}

std::scoped\_lock lock(); ```

std::vector v1();

std::vector v2{};

### 6.25.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects, C++ software developers can:

— Use the avoidance mechanisms of ISO/IEC 24772-1 clause 6.25.5.

* Simplify expressions to aid in code readability and help future maintainers understand the intent and nuances of the code. For example,
  + by using parentheses around sub-expressions to limit precedence confusion,
  + by extracting intermediate results into const variables with a meaningful name, or
  + by extracting a subexpression into a function with a meaningful name.
* Avoid assignments embedded within other statements and expressions.
* Spell unary operators (e.g., -) with a leading blank in expressions to avoid them being misread as combined operators.
* Avoid the use of unary plus, since it is almost always a no-op for built-in types.
* Avoid Boolean operators (&&, ||, !) with non-bool operands, e.g., operands of numeric types.
* Avoid bit operators (&, |, ~, <<, >> ) with anything except operands of non-bool unsigned types.
* Consider using alternative tokens for the logical operators, such as and, or, and not, and comparison operators such as equal and not\_eq.
* If your code structure requires an empty statement ; use an empty code block instead {}.
* Prohibit conflicting side-effects in sub-expressions.
* Avoid defining semantics of overloaded operators to deviate from the semantics of these operators for the built-in types.
* Prefer defaulted and synthesized comparison operators over individual overloads to ensure that all of the related comparison operators behave consistently.

## 6.26 Dead and Deactivated Code [XYQ]

### 6.26.1 Applicability to language

The vulnerability as documented in ISO/IEC 24772-1 clause 6.26 exists in C++.

The language mechanisms around templates and overload resolution can require definitions to exist that are not part of the executable program. But the mechanisms at compile time guarantee that the corresponding code never becomes part of the executable program. However, a programmer might be unaware of all details with respect to the language mechnisms and thus make subtle errors leading to code selected for the executable program that was unintended.

If there is code that was once needed or might be needed in the future, programmers might opt to comment or use preprocessor conditional compilation to exclude such parts. The latter might even be confusing, because an intentionally undefined macro might be defined for a specific compilation outside of the program source text. Modern version control systems are better places to keep unused code in a revision or branch and ressurect it if needed through a merge.

### 6.26.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects, C++ software developers can:

* Use the avoidance mechanisms of ISO/IEC 24772-1 clause 6.26.5.
* Use IDEs or similar tools that visualize the code that gets selected for the executable program.
* Use modern version control to manage unused code.

## 6.27 Switch Statements and Static Analysis [CLL]

### 6.27.1 Applicability to language

The vulnerability as described in ISO/IEC TR 24772-1:2019 clause 6.27 exists in C++.

Because of the way in which the switch-case statement in C++ is structured, it can be relatively easy to unintentionally omit the break statement between cases causing unintended execution of statements for some cases.

The switch statement has the form:

int abc = someExpression();  
 /\* … \*/  
 switch (abc) {  
 case 1:  
 sval = “a”;  
 break;  
 case 2:  
 sval = “b”;  
 break;  
 case 3:  
 sval = “c”;  
 break;  
 default:  
 throw SomeException();  
 }

If there isn’t a default case and the switched expression doesn’t match any of the cases, then control simply shifts to the next statement after the switch statement block. Unintentionally omitting a break statement between two cases will cause subsequent cases to be executed until a break or the end of the switch block is reached. This could cause unexpected results.

The attribute [[fallthrough]] expresses the programmer’s intent that the code where it is placed is intended to fall through. If this attribute is not used, compilers typically diagnose the absence of a break statement.

### 6.27.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects, C++ software developers can:

* Use the avoidance mechanisms of ISO/IEC 24772-1 clause 6.27.5
* Use [[fallthrough]] wherever fall-through is intended.
* Terminate every case with either a flow control transfer or [[fallthrough]] as illustrated in the following example:
* ```{.cpp}  
  int i;  
  . . .  
  switch (i) {  
   case 1:   
   [[fallthrough]]; // documents the intended fallthrough.  
   case 2:  
   i++;   
   break;  
   case 3:   
   j++;   
   [[fallthrough]]; // documents the intended fallthrough.  
   case 4: //other code  
   return 42;  
   default: throw CaseNotFound();   
   }  
  ```
* Adopt a style that permits your language processor and analysis tools to verify that all cases are covered. Where this is not possible, use a default clause that diagnoses the error.

See also the C++ Core Guidelines ES.78

## 6.28 Demarcation of Control Flow [EOJ]

### 6.28.1 Applicability to language

The vulnerability as described in ISO/IEC TR 24772-1:2019 clause 6.28 exists in C++.

C++ lacks a keyword to be used as an explicit terminator. Therefore, it may not be readily apparent which statements are part of a loop construct or an if statement.

Consider the following sections of code:

int foo(int a, const int \*b) {  
 int i=0;  
 // . . .  
 a = 0;  
 for (i=0; i<10; i++); // notice the ';' !!  
 {  
 a = a + b[i];  
 }  
 int c = 0;  
 int x = 0;  
 for (int j=0; j<10; j++)  
 c = c + b[j];  
 x += c;   
 }

At first it may appear that, after the first loop, a will be a sum of the numbers b[0] to b[9]. However, even though the code is laid out so that the a = a + b[i] code appears to be within the for loop, the “;” at the end of the for statement causes the loop to be on a null statement (the “;”) and the

a = a + b[i];

statement to only be executed once. Similarly, the indentation leads us to believe that that assignment to x is part of the second loop, but it is not. These mistakes may be readily apparent during development or testing. More subtle cases may not be as readily apparent leading to unexpected results.

if statements in C++ are also susceptible to control flow problems since there isn’t a requirement in C++ for there to be an else statement for every if statement. An else statement in C++ always belong to the most recent if statement without an else. However, the situation could occur where it is not readily apparent to which if statement an else belongs due to the way the code is indented or aligned.

Similar issues arise for if-statements, particularly during maintenance, for example:

```{.cpp}  
int a,b,i;  
// . . .   
if (i == 10){  
 a = 5;   
 b = 10; // added later, but correct since within the {…}  
 }  
 else   
 a = 10;   
 b = 5; // added later, intended to be part   
 // of the else clause  
```

If the assignments to b were added later and were expected to be part of each if and else clause (they are indented as such), the above code is incorrect: the assignment to b that was intended to be in the else clause is unconditionally executed.

### 6.28.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects, C++ software developers can:

* Use the avoidance mechanisms of ISO/IEC 24772-1 clause 6.28.5.
* Enclose the bodies of if, else, while, for, and similar in braces. This will reduce confusion and potential problems when modifying the software.
* Declare loop variables in the initializer of the loop statement
* Prefer the standard library algorithms over hand-crafted loops.
* Consider the use of source code auto-formatters before code reviews.

See also the C++ Core Guidelines ES.85, ES.71, ES.74, ES.1 and ES.2

## 6.29 Loop Control Variables [TEX]

### 6.29.1 Applicability to language

The vulnerability as described in ISO/IEC TR 24772-1:2019 clause 6.29 exists in C++.

C++ allows the modification of loop control variables within non range-based loops. This is usually not considered good programming practice as it can cause unexpected problems. The flexibility of C++ expects the programmer to use this capability responsibly.

Since the modification of a loop control variable within a loop is infrequently encountered, reviewers of C++ code may not expect it and hence miss noticing the modification. Modifying the loop control variable can cause unexpected results if not carefully done. In C++, the following is semantically correct, but is error-prone:

int a;  
for (int i = 1; i < 10; i++){  
 ...  
 if (a > 7)  
 i = 10;  
 ...  
}

which will cause the for loop to exit once a is greater than 7 regardless of the number of iterations that have occurred.

for (int i : std::ranges::iota\_view{1,10})  
{  
 if (a > 7) {  
 i = 10; // This changes the local variable for this loop iteration's execution  
 // but subsequent iterations are not affected  
 }  
 }  
   
for (int const i : std::ranges::iota\_view{1,10})  
{  
 if (a > 7) {  
 i = 10; // This is now illegal since the 'const int' prevents assign  
 }  
 }

The range for example immediately above does not have the vulnerability of the C-like for loop above.

for (int i=1; i < 10; ++i) ...

In a range-based for loop, the control variable is not available.

std::array a {3, 1, 4, 1, 5};  
for (auto const x : a) {  
 std::cout << x << '\n';  
}

### 6.29.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects, C++ software developers can:

* Use the avoidance mechanisms of ISO/IEC 24772-1 clause 6.29.5.
* Do not modify a loop control variable within a loop.
* Use a range-for loop in preference to general loops.
* Alternatively, use standard library generic algorithm functions like copy, reduce, transform, inner\_product, etc. in preference to general loops.

Note: See also the C++ Core Guidelines ES.71, ES.86.

## 6.30 Off-by-one Error [XZH]

### 6.30.1 Applicability to language

The vulnerability as described in ISO/IEC TR 24772-1:2019 clause 6.30 exists in C++.

Arrays are a common place for off by one errors to manifest. In C++, arrays are indexed starting at 0, causing the common mistake of looping from 0 to the size of the array as in:

int foo() {  
 int a[10];  
 int i;  
 for (i=0, i<=10, i++)  
 ...  
 return (0);  
}

C++ mitigates the issue of sentinel values in strings document in ISO/IEC 24772-1 clause 6.30 by providing the string class and the string\_view class.

C++ does not flag accesses outside of array bounds, so an off by one error may not be as detectable in C++ as in some other languages. Several good and freely available tools can be used to help detect accesses beyond the bounds of arrays that are caused by an off by one error. However, such tools will not help in the case where only a portion of the array is used, and the access is still within the bounds of the array.

C++ mitigates these issues by providing

* Range-based for loops
* std algorithms
* Iterator style loops terminated by !=
* Container classes
* gsl::span (soon to be std::span)

### 6.30.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects, C++ software developers can:

* Use the avoidance mechanisms of ISO/IEC 24772-1 clause 6.30.5.
* Use careful programming, testing of border conditions, and static analysis tools to detect off-by-one errors in C++.
* Use range-based for loops, std algorithms, iterator style loops terminated by !=, or container classes in preference to C-style arrays and structures.

See also the C++ Core guidelines ES.1, ES.42, ES.71, SL.con.3 (more to come)

## 6.31 Structured Programming [EWD]

### 6.31.1 Applicability to language

The vulnerability as described in ISO/IEC TR 24772-1:2019 clause 6.31 exists in C++.

It is as easy to write structured programs in C++ as it is not to. C++ contains the goto statement, which can create unstructured code. It also has continue, break, and return that can create a complicated control flow, when used in an undisciplined manner. Spaghetti code can be more difficult for static analyzers to analyze and is sometimes used on purpose to intentionally obfuscate the functionality of software. Code that has been modified multiple times by an assortment of programmers to add or remove functionality or to fix problems can be prone to become unstructured.

Because unstructured code in can cause problems for analyzers, both automated and human, of code, problems with the code may not be detected as readily or at all as would be the case if the software was written in a structured manner.

In C++, the break and continue operations only act on the innermost loop. At times, escape from nested loops is required. In such cases, the use of goto may be simpler and easier to verify than a series of tests with break and/or continue operations.

The setjmp macro sets the current execution context into a variable, which can be use later to return to that current context using longjmp call. These calls originated from the C standard library to mimic goto across the call stack. They do not support the relevant additions to C++ such as destructors for automatic objects, exceptions, and concurrency, and hence are incompatible with modern C++ programming.

A coroutine is a function that can suspend execution for later resumption (optional).

### 6.31.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects, C++ software developers can:

* Use the avoidance mechanisms of ISO/IEC 24772-1 clause 6.31.5.
* Write clear and concise structured code to make code as understandable as possible.
* Avoid the use of setjmp and longjmp
* Consider the use of coroutines as an alternative to goto and setjmp/longjmp

See also the C++ Core guidelines ES.76, ES.77, SL.C.1

## 6.32 Passing Parameters and Return Values [CSJ]

### 6.32.1 Applicability to language

The vulnerability as described in ISO/IEC 24772-1:2024 6.32 exists in C++. However, the language also provides appropriate mitigation.

C++ provides both *call by copy* (aka *call by value*) and *call by reference* parameter passing. The argument is evaluated to initialize the formal parameter (in the first case) or bound to the formal parameter (in the second case) of the function that is being called. A formal parameter behaves like a local variable, however, changes to a non-const reference parameter affect the bound object.

The rich type system of C++ allows types that when passed by value still have *call by reference* semantics, for example, pointer types, std::reference\_wrapper, or class types with pointer or reference member variables.

C++ assumes that pointer or reference parameters of different types never alias, even if the underlying object representations are identical, i.e., for a function declared as void f(int \*pi, long \*pl) the compiler will assume that pi and pl always refer to different objects, even if sizeof(int) == sizeof(long). Two parameters may refer to the same object if they have pointer or reference type, and the target types are the same or related. This means, aliasing between reference parameters or with a reference result needs to be taken into account in user code. For example, in an assignment expression the left and right hand side can refer to the same object. This implies that user-defined assignment operators must take precautions against self-assignment or document that it is forbidden.

Modern C++ ensures that in many cases the need for and overhead of copying value arguments or results is elided by the compiler, especially from temporary objects.

The use of const lvalue-reference parameters combines the efficiency of *call by reference* with the guarantee that the underlying input parameter is not changed (marking it as an *in* parameter). A non-const reference parameter must be considered an *inout* parameter. Rvalue-reference parameters are *inout* parameters that allow *transfer-of-ownership* semantics. At their call site it is best to assume that the argument object is in an indeterminate state and has to be reassigned before subsequent use. There is no language mechanism for marking *out* parameters, one would use the return mechanism. Instead of multiple *out* parameters a struct, std::pair, or std::tuple can be used as a return type and eventually decomposed at the call site to its constituents via a *structured binding*.

Member functions take the \*this object as an implicit reference parameter. The kind of reference can be specified through qualification of the member function. However, in addition to lvalue-reference, const-lvalue-reference, and rvalue-reference qualification, there exists an oddity with respect to normal reference parameters:

* without a qualification a member function can be called on rvalues and non-const lvalues, and
* with a const-qualification a member function can be called on const lvalues and rvalues.

This means, unqualified member functions are callable on temporaries (rvalues) and thus can have side effects, but also can return an lvalue-reference to said temporary by returning \*this (or members of \*this), which can lead to dangling if such a reference is used beyond the expression of the function call returning it. For example, the compiler-provided assignment operators of a class are unqualified member functions that return an lvalue-reference to \*this.

Rvalue-reference parameters in a context where their actual type is deduced from the call site, are called *forwarding references*. A forwarding references will either be deduced to an lvalue-reference or an rvalue-reference depending on the argument at the call site.

Aliasing is expected and allowed in some cases, such as:

- Assignment and compound assignment operators: the right parameter may alias the left parameter. The function result always refers to the left parameter unless overloaded differently. In the case of self-assignment the stored value should not change.  
  
- Functions that `swap`{.cpp} their parameters: The two parameters to be swapped may refer to the same object.  
  
- Shift operators used for input and output: the result always refers to the left parameter.  
  
- Prefix increment and decrement operators: the result always refers to the parameter.

The C++ preprocessor macros use a *call by name* parameter passing; a call to the macro replaces the macro by the body of the macro. This is called *macro expansion*. Macro expansion is applied to the program source text and amounts to the substitution of the formal parameters with the actual parameter expressions. Formal parameters are often parenthesized to avoid syntax issues after the expansion. Call by name parameter passing reevaluates the actual parameter expression each time the formal parameter is read.

### 6.32.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects, C++ software developers can:

* Apply the avoidance mechanisms of ISO/IEC 24772-1 6.32.5.
* Prefer pass-by-value and pass-by-const-reference over other means of parameter declaration.
* Prefer reference parameters over corresponding pointer parameters.
* Use pass by non-const reference only when side-effects on the argument are intended (*inout*-parameter).
* Unless the moved-from state is well-defined, always reassign a variable passed by rvalue-reference before subsequent use after the call.
* Use forwarding reference parameters only, if perfect forwarding is needed in the body of a function template or generic lambda.
* Define member functions with the appropriate lvalue-ref-qualification when they have side effects on \*this or return a pointer-like object referring to \*this or its members.
* Either delete the rvalue-ref-qualified overload or define it as returning by value, when a const-ref-qualified overload of a member function exists that returns a pointer-like object referring to \*this or its members.
* Document the referents of pointer-like objects created by or modified by a function if any pointer-like object outlives the invocation of that function.
* Document any allowable aliasing between the referents of function parameters of pointer-like type. Absent such documentation, avoid passing aliased parameters.
* Do not use function-like macros. If used, surround all macro parameters in the replacement text with parentheses.

## 6.33 Dangling References to Stack Frames [DCM]

### 6.33.1 Applicability to language

The vulnerability as expressed in ISO/IEC IS 24772-1:2024 and ISO/IEC TR 24772-3:2020 C exists in C++ by indirect access to variables with automatic storage duration or to temporary objects.

The lifetime model of C++ makes it undefined behaviour [[EWF]](#EWF) to access an object outside of its lifetime. This results in undefined behavior, when an object access is attempted after its destruction. C++ provides a rich set of *pointer-like* types whose values may refer to temporaries or variables with automatic storage duration and can dangle (see Subclause [[XYK]](#XYK)).

A C++ class type with a *pointer-like* member will behave as a *pointer-like* type, unless the class itself manages the lifetime of the object referred to by its member.

In general, any caller storing the *pointer-like* object returned from a function call risks dangling; such situations require thorough lifetime analysis to ensure that access via the *pointer-like* object doesn’t dangle.

The efficiency of return-by-value, copy-elision, and move-semantics as specified by C++ reduces the incentive to return a *pointer-like* type from a function or bind a temporary to a local reference.

The lifetime of a temporary object usually ends at the end of a *full expression* where it was created. Dangling can occur, when an expression including the creation of a temporary object results in a *pointer-like* value referring to the temporary object. For example, std::max returns the const-reference given as parameter, which might be bound to a temporary argument:

int g(int i){  
 int const &m = std::max(i,20);  
 return m; // access dangling reference to temporary if i < 20  
}

In some situations binding a reference to a temporary will extend the lifetime of the temporary.

This lifetime extension is not transitive across function calls, therefore, changes in the code, such as replacing a data member access with an accessor member function, can silently lead to dangling in such lifetime-extension situations.

struct A{  
 int a;  
 int const &getA(){return a;}  
};  
void h(){  
 int && ra = A{42}.a; // lifetime extended  
 int const & cra = A{42}.getA(); // dangling  
}

The range-based for statement contains a subtle situation with lifetime extension.  
A temporary in the range expression will have its lifetime extended, unless it is accessed indirectly. As a mitigation C++ permits the creation of a variable for such situations that has the scope of the range-for loop, as shown in the following example:

extern std::vector<std::string> make(); // creates a vector  
  
for(char c : make().front()) { // attempt to iterate over first string in vector  
 // vector and thus contained string is already destroyed before C++23  
}  
  
for(auto range = make().front(); char c : range){ // mitigation, create a variable for the range to be iterated over  
 // string to be iterated over remaings valid throughout  
}

This issue is no longer present from C++23 onwards, as temporaries within the for-range-initializer are lifetime extended until the end of the statement.

Returning a *pointer-like* object from a function is problematic, if the return value refers to a temporary or an object with automatic storage duration, either directly or indirectly. The following example show different situations with this problem:

int \*bad\_pointer() {  
 int a = 0;  
 return &a; // Returning the address of a local variable "a".  
 }  
  
int& bad\_reference(int b) {  
 return b; // Returning a reference to a local (parameter) variable "b" .  
 }  
  
std::array<int,3>::iterator bad\_iterator() {  
 std::array<int,3> c = { 1, 2, 3 };  
 return c.begin();  
 // Returning an iterator that refers the first element of the local array "c".  
}  
  
auto bad\_lambda() {  
 int d = 0;  
 return [&] { return d = 1; };  
 // Returning a lambda that captures local variable "d" by reference  
 // and thus indirectly returns a reference to the local variable  
}  
decltype(auto) bad\_assign(){ // deduces: std::string &  
 return std::string{} = "hello\n"s;  
 // Returns reference to temporary object returned from copy-assignment operator  
}  
  
void erroneous\_use() {  
 std::cout << \*bad\_pointer();  
 std::cout << bad\_reference(42);  
 std::cout << \*bad\_iterator();  
 std::cout << bad\_lambda()();  
 std::cout << bad\_assign();  
 }

In the examples above, the function bad\_assign returns a std::string & that was itself returned from the copy-assignement operator of std::string. Such an assignement operator (including the compiler-provied ones) can be called with a temporary as its left-hand operand, because it is an unqualified member function (for historical reasons).

Dangling may occur by calling a member function on a temporary that returns a *pointer-like* object referring to \*this, a sub-object of \*this, or an object managed by \*this. This can be prevented by - For a non-const member function: adding an lvalue ref-qualification (&), - For a const member function: adding an lvalue ref-qualification (const &) **and** declaring an rvalue ref-qualified overload (&&) either defined as =delete or declared to return by value.

In the following example, class nta declares its copy assignment with lvalue ref-qualification to avoid the situation created in the example function bad\_assign:

struct ta{}; // default allows assignment to temporary  
struct nta{  
nta & operator=(nta const &) & = default; // lvalue-ref qualified  
};  
ta & check\_ta(){  
 return ta{} = ta{}; // returns dangling reference to temporary  
}  
nta & check\_nta(){  
 return nta{} = nta{}; // won't compile  
};

Referring to a variable with automatic storage duration from a *pointer-like* variable with static or thead-local storage duration usually means dangling, when the indirect access happens.

int const init{42};  
std::reference\_wrapper<int const> bad\_ref = init; // static storage duration  
void bad\_global\_assign(){  
 if (bad\_ref == 42){ // undefined behavior on 2nd call  
 int local{44};  
 bad\_ref = local; // Any further access of bad\_ref dangles  
 }  
}

A class type with *pointer-like* members can lead to dangling when those members refer to constructor arguments.

struct X{  
int const &rci;  
X(int i):rci{i}{} // No lifetime extension of parameter object by binding reference to it  
};

Similarly, in the following example the vulnerability exists in the conversion operator string\_view() of std::string, that returns a *pointer-like* type from a member function callable on a temporary object.

std::string\_view bad\_var("a string"s); // dangling view on temporary string object

A C++-specific way of causing dangling\_references to the stack is by means of a placement new expression (see 7.4 Placement new issues [XXX]).

### 6.33.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects, C++ software developers can:

* Prefer value types, pass-by-value, and return-by-value over *pointer-like* types and passing or returning *pointer-like* objects.
* Prohibit the following uses of a *pointer-like* value referring to a variable with automatic storage duration or referring to a temporary object:
  + Returning the *pointer-like* value from a function;
  + Assigning the *pointer-like* value to a *pointer-like* variable with a greater lifetime, such as a variable with static or thread storage duration, or initializing a *pointer-like* member variable from a constructor value parameter.
* Avoid capturing by reference in a lambda that will be used non-locally, that is
  + returned,
  + passed to another thread or coroutine, or
  + stored in a variable with a longer lifetime than the captured variables.
* Avoid relying on lifetime extension of temporaries by binding them to named references; use (local) variables instead.
* Show that the *range-based for* dangling vulnerability does not apply or take steps to avoid it, for example, use a variable representing the range and not an expression that yields a reference to a temporary.
* Consider making member functions ref-qualified, that return *pointer-like* types to members or objects managed by the class.  
  If required, provide an rvalue-ref-qualified overload that either returns a copy by value, or is defined as =delete to prevent calling it on a temporary.
* Perform lifetime analysis when using a *pointer-like* object beyond the expression that created it and ensure it is not used in a dangling situation.
* Employ static and dynamic analysis tools to detect dangling *pointer-like* objects.

## 6.34 Subprogram Signature Mismatch [OTR]

### 6.34.1 Applicability to language

The vulnerabilities documented in ISO IEC 24772-1 6.34 is applicable to C++ as described below.

In general in C++, there must be a match between the number of parameters in a function call and the number of arguments in the function definition. For issues related to macro signatures, see subclause [Pre-processor directives[NMP]](NMP).

The number of arguments can be different to the number of parameters in a function where: - a function template includes a *function parameter pack*, or - a function parameter includes a default argument, or - a function parameter-declaration-clause ends with an ellipsis, f(\...).

Calling a function template with a *function parameter pack* results in a specialization of the function with the parameter types matching the corresponding argument types.

The compiler will ensure for variadic templates that the type and number of arguments is correct.

A call to a function with default arguments can provide fewer arguments than parameters as long as the parameters for which no explicit argument is provided include a default argument.

Where a function parameter-declaration-clause ends with an ellipsis, additional arguments can be accessed through the mechanisms provided by <cstdarg>.  
No information about the number or types of the parameters is supplied by the compiler. The use of this feature outside of special situations can be the basis for vulnerabilities. and undefined behavior [[EWF]](#EWF) can arise, for example:

* Arguments with integral or enumeration type that are passed to an ellipsis parameter will undergo integral promotions, and so the type of the parameter is int and not short. The use of short with va\_arg is therefore invalid.

#include <cstdarg>  
  
void f1 (int cnt, ...)  
{  
 va\_list ap;  
 va\_start (ap, cnt);  
  
 short i = va\_arg (ap, short); // Invalid type  
  
 va\_end(ap);  
}

* In the following case, f1 assumes that there will be two parameters after the named parameter.

#include <cstdarg>  
  
void f1 (int cnt, ...)  
{  
 va\_list ap;  
 va\_start (ap, cnt);  
  
 int i = va\_arg (ap, int);  
 int j = va\_arg (ap, int);  
  
 va\_end(ap);  
}  
  
void f2 ()  
{  
 f1 (1, 2, 3); // OK  
 f1 (1, 2); // results in undefined behaviour  
}

These issues cannot occur where default arguments or variadic function templates are used.

The C++ Name mangling ensures that function signatures match accross translation units.

This does not apply to other mangling schemes. For example, parameters do not form part of the mangled name for functions declared with the extern "C" linkage specification. Thus such a function can be invoked with incorrect parameter types due to an incorrect redeclaration of the function:

// library.cc  
extern "C" void foo (unsigned, unsigned)  
{  
 // ...  
}

// main.cc  
  
extern "C" void foo (unsigned);  
  
int main ()  
{  
 foo (0xffffffff); // Calling function that is  
 // defined to take 2 parameters  
}

### 6.34.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects, C++ software developers can:

* Use the avoidance mechanisms of ISO/IEC TR 24772-1:2019 clause 6.34.5.
* Avoid using functionality from <cstdarg>; use variadic function templates instead.
* When using functions declared with extern "C" linkage specifications, use static analysis tools to ensure that all such declarations of equally named functions have identical signatures.

Note: See also C++ Core Guidelines F.55.

## 6.35 Recursion [GDL]

### 6.35.1 Applicability to language

C++ permits recursion, hence is subject to the problems described in ISO/IEC 24772-1 clause 6.35.

C++ allows recursive constexpr functions and consteval functions that are evaluated at compile time where such calls don’t contribute to the vulnerability.

### 6.35.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects, C++ software developers can follow the avoidance mechanisms of ISO/IEC 24772-1 clause 6.35.5.

## 6.36 Ignored Error Status and Unhandled Exceptions [OYB]

### 6.36.1 Applicability to language

The vulnerabilities described in ISO/IEC 24772-1:2019 clause 6.36 exist in C++, however, C++ provides a mitigation.

C++ includes the C library, especially the header <cerrno> and thus shares C’s issues with the global error-reporting variable errno. See ISO/IEC 24772-3 clause 6.36 for details and guidance.

In addition to errno some C++ library features expose error conditions indirectly via a side-effect on the object the operation failed with or via side-effect on a reference parameter. For example, input stream objects will go into a fail state, when formatted input cannot be performed. Without resetting that fail-state of a stream, further input will continue to fail, that can cause further failures when a failure is ignored. Streams provide a non-default mode to throw exceptions on failure. Another example of error reporting via a side-effect is the filesystem library that provides overloads that take a non-const reference of std::error\_code.

In general, reporting errors as side-effects, in the worst case via a global state, is too easy to accidentally ignore by developers, leading to further consistency problems in the continued execution of the program.

By default, C++ has the C weakness of permitting the call to a function that returns an error code without capturing the return value in a variable.

errnum foo( int a, int b);  
. . .  
foo(x, y); // failure to capture the return error code.

C++ offers as a mitigating mechanism the [[nodiscard]] attribute. This attribute indicates that the function result must not be discarded. Ignoring the result of a function marked [[nodiscard]] causes a compiler warning.

[[nodiscard]] errnum foo( int a, int b);  
. . .  
foo(x, y); // compile error.  
  
if( auto e = foo(a,b); e == 0) { // no compile error  
// success  
}  
else {  
// handle errors  
}

In addition, the C++ library provides mechanism to extend the return type of a function with extra values for denoting an error. The simplest case is std::optional<T> that extends T with an “empty” state. Callers must check the result of functions returning an optional for the empty state, before accessing its value. This increases the chances that a reported error is detected by the caller. If additional information of the error has to be returned to the caller std::expected<T,AnErrorCode> or alternatively std::variant<T,AnErrorCode> can be used.

C++ offers a set of library-defined exceptions for error conditions that may be detected by checks that are performed by the standard library. In addition, the programmer may define exceptions that are appropriate for their application. These exceptions are handled using an exception handler. Exceptions may be handled in the environment where the exception occurs or may be propagated out to an enclosing scope. Exceptions that are never handled in the program result in abnormal termination of the application. In this case, it is implementation-defined whether the destruction of local objects (stack unwinding) occurs. An unhandled exception that occurs in a thread also results in the abnormal termination of the application. See 6.62 Concurrency - Premature Termination [CGS] for issues related to thread or process termination. An exception propagating out of a coroutine causes the coroutine to end in an unresumable state and the exception is not further propagated.

### 6.36.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects, C++ developers can:

* Follow the avoidance mechanisms of ISO/IEC 24772-1 subclause 6.36.5;
* Develop an error management strategy, including error-detection, error-reporting, and error-handling,that, based on context, selects the most appropriate mechanism(s) from the different ones provided by C++;
* Develop an error management strategy, including error-detection, error-reporting, and error-handling that selects the most appropriate mechanism(s), depending on the context, from the different ones provided by C++;
* Prohibit error handling based on unsynchronized global state;
* Avoid the use of errno, but if it cannot be avoided, follow the avoidance mechanisms of ISO/IEC TR 24772-3 subclause 6.36.2;
* Use static analysis tools to detect and report missing or ineffective error detection or handling;
* Use [[nodiscard]] to prevent callers from ignoring error values and configure your compiler to report violations as errors, such as -Werror;
* Use a try-catch function body for main or any thread-entry function with a catch(...) handler to guarantee stack unwinding, which guarantees the destruction of all local objects and permits control of the ultimate termination of the thread or program;
* Use a try-catch body for a coroutine with a catch handler to prevent silent termination of the coroutine in the case of an exception, or alternatively use a coroutine type that properly supports the case where the coroutine ends with an exception;
* Consider controlled termination as a last resort strategy for main or for noexcept functions;
* Consider the use of std::exception\_ptr or an object encapsulating its mechanics like std::future to transport an exception from one thread to another thread for further processing.

## 6.37 Type-breaking Reinterpretation of Data [AMV]

### 6.37.1 Applicability to language

The vulnerability as documented in ISO IEC 24772-1 clause 6.37 applies to C++. The language mechanisms where type-breaking reinterpretation of data can happen involve unions and std::bit\_cast. For type-breaking reinterpretation involving pointers or references see [6.11 Pointer Type Conversions [HFC]](#HFC).

In C++, the issue of casting vulnerabilities are mitigated by the fact that the C++ named casts, i.e., reinterpret\_cast, const\_cast, static\_cast, dynamic\_cast, and std::bit\_cast, are easily identified, e.g., by text search for more careful review. C-style casts do not share this property and therefore should be prohibited. In addition, C++ named casts include some compile-time checks, and in the case of dynamic\_cast run-time checks, that help avoid some but not all vulnerabilities.

Of the available "\_cast" operations, only std::bit\_cast provides reinterpretation of data values, and only reinterpret\_cast allow the reintepretation of data as a different type within limits. For uses of reinterpret\_cast see [6.11 Pointer Type Conversions [HFC]](#HFC). The named casts static\_cast, dynamic\_cast, and const\_cast perform type conversions and not reintepretation of bits and thus don’t have this vulnerability but are subject to potential conversion errors (see [6.6 Conversion Errors [FLC]](#FLC)).

Reading a union member that was not previously written is undefined behaviour [[EWF]](#EWF) except for a few cases described by ISO/IEC 14882:2020 clause [class.mem.general]. The reinterpretation of data values via different union members that is common practice in C is undefined behaviour in C++. The type std::variant provides a similar mechanism to union but prevents reading an inactive member.

If there is no value of the target type corresponding to the bit representation of the source type’s value, using std::bit\_cast is undefined behaviour, such as in the following example.

enum E { one, two, four=4 };  
constexpr auto x = std::bit\_cast<E>(42); // 42 is not representable by E

C++ also provides <type\_traits>, such as std::is\_layout\_compatible, that can be used to ensure the legality of a specific std::bit\_cast.

A legacy means of “bit casting” is the use of memcpy to transfer the bytes of an object’s representation to another object of a different type. Except for cases that are well-defined with std::bit\_cast, such use of memcpy to reinterpret data values is undefined behaviour.

### 6.37.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects, C++ software developers can:

* follow the avoidance mechanisms of ISO/IEC 24772-1 clause 6.37.5, and clauses [6.6 Conversion Errors [FLC]](#FLC) and [6.11 Pointer Type Conversions [HFC]](#HFC) of this document.
* Prohibit casts that reinterpret data values.
* Whenever possible use std::variant or a similar safe union type instead of union.
* Prefer std::bit\_cast over memcpy when reinterpreting the bit-representation of an object to a different type is unavoidable.
* Use std::is\_layout\_compatible, and std::is\_trivially\_copyable and other <type\_traits> to mitigate incorrect usage of std::bit\_cast.
* Ensure through static analysis that no accidental type-breaking reinterpretation of data values occur.

## 6.38 Deep vs. Shallow Copying [YAN]

### 6.38.1 Applicability to Language

The vulnerability described in ISO/IEC TR 24772-1:2019 clause 6.38 exists in C++ and only arises in C++ when there is a mismatch between the type’s copy semantics and the programmer’s intent.

On the language level, reference semantics, that can lead to shallow copies, usually requires the use of pointer or reference types, however, an integral type can also have reference semantics, for example, when it is used as an index or as an operating system handle.

Such types with reference semantics are also called *pointer-like types* and they will suffer from the aliasing problematic of this vulnerability and additionally from potential dangling due to expired lifetime of referred objects (see [[XYK]](#XYK)).

In general, relationship types with an immutable referent, such as a const-reference, do not suffer the deep vs. shallow copying semantics problem, unless mixed with relationship types with a mutable reference to the same object. However, the lifetime of a const referent is still an issue to manage (see [[XYK]](#XYK)).

The standard library type std::shared\_ptr<T> has shallow copy semantics when the managed type is non-const, but in contrast to other relationship types guarantees the lifetime of the referent.

Class types that have *relationship type* members will become *relationship types* themselves, unless the class provides deep copy semantics or disables it and manages the lifetime of the referred object (*manager type*). Such *relationship types* and *manager types* will refer to their referred/managed resources via a data member with reference semantics.

A *manager type* defines a non-empty, non-deleted destructor in addition to providing appropriate copy and move operations. Examples of a *manager types* are the standard library container types such as std::vector that use pointers to the allocated space of their elements and copying a vector will also copy all contained elements not just the pointers. This management is achieved by replacing the compiler-provided copy-constructor and copy-assignment operator with implementations providing value semantics that perform the deep copy (*general manager*). An alternative to potentially expensive deep copies for *manager types* is the prevention of copying, either by defining move operations that transfer the ownership of a managed resources, like std::unique\_ptr does (*unique manager*), or by preventing both copy and move operations (*scoped manager*), for example, by defining the move-assignment operator as deleted.

Without such replacement of copy and move operations a class type with *relationship type* members suffers from the potential confusion due to shallow copies. For example, the standard library types std::span, std::string\_view, iterators, and the views of the standard ranges library are relationship types. Care must be taken to not only understand implications of their shallow copy semantics, but also about their validity depending on the lifetime of the referred ranges.

Using *relationship types* as function parameter types is usually safe, because language semantics guarantee the lifetime of parameter objects. Exceptions exists for thread functions and coroutines, where the initial calling context is not guaranteed to exist when parameters of relationship type are accessed.

Returning a relationship type from a function can be problematic, unless the lifetime of the referred object is clear. For example, returning a reference to a local variable will return a dangling reference (see [[XYK]](#XYK) and [[XYH]](#XYH)).

See also Core Guidelines C.20, C.22, C.32, C.67.

### 6.38.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects, C++ software developers can:

* Follow the avoidance mechanisms of ISO/IEC 24772-1 clause 6.38.5.
* Use and create value types.
* Consider *pointer-like types* only when the lifetime of the referent can be ensured, and
  + when reference semantics is required, e.g., for side effects, or
  + when copying is prohibitively expensive.
* Be aware of *pointer-like types* and document the reference semantics of user-defined class types.
* For *manager types*, define copy constructor, copy assignment operator, move constructor, move assignment operator, and their destructor appropriately.
* Prefer value parameters to thread functions and coroutines.

## 6.39 Memory Leak and Heap Fragmentation [XYL]

### 6.39.1 Applicability to language

The memory leak vulnerability documented in ISO/IEC 24772-1:2024 clause 6.39 exists in C++, unless the programmer takes steps to avoid it. A leak occurs when an object whose lifetime (see 4.4) has not ended but there are no more variables, pointers or references to that object.

See ISO/IEC TR 24772-3 for the vulnerabilities associated with the C functions malloc(), calloc(), realloc() and free().

C++ has an additional vulnerability in that it provides multiple alternatives for allocation and deallocation.

Failing to match the deallocation to the corresponding allocation results in undefined behaviour [[EWF]](#EWF). For example, if an array new[] expression was used to allocate and create an array then array delete[] must be used for its destruction and release to avoid undefined behaviour.

C++ destructors allow *scope-based resource management* (RAII) (Note: - put in Terms and definitions) which mitigates memory leaks by automatically calling the destructor when the lifetime of an object ends, thereby providing the opportunity to release their memory. The standard library provides container classes that follow the *scope-based resource management* idiom for the management of memory resources. Additionally, the standard library provides the class templates std::unique\_ptr and std::shared\_ptr for managing heap-allocated objects.

Ignoring issues with constructors throwing exceptions during heap allocation with a new expression can cause leaks. Using *smart pointers* through the factory functions std::make\_unique() and std::make\_shared() assists in avoiding memory leaks.

If using functions that manage memory using the C library mechanisms is unavoidable, wrapping such a pointer immediately into a specialization of std::unique\_ptr<> that uses free() in its deleter object ensures that memory is correctly released when the unique\_ptr is destroyed, for example:

struct free\_deleter{  
 template <typename T>  
 void operator()(T \*p) const {  
 std::free(const\_cast<std::remove\_const\_t<T>\*>(p));  
 }  
};  
template <typename T>  
using unique\_C\_ptr=std::unique\_ptr<T,free\_deleter>;  
//...  
// abi::\_\_cxa\_demangle() returns a pointer to be released with free()  
inline auto plain\_demangle(char const \*name){  
 unique\_C\_ptr<char const> result {abi::\_\_cxa\_demangle(name,0,0,0)};  
 return result;  
}

Modern C++ provides a class hierarchy, std::pmr::memory\_resource, that contains advanced allocation strategies and removes the need for explicit allocation/deallocation from the programmer, as shown below.

#include <array>  
#include <cstdlib>  
#include <iostream>  
#include <memory\_resource>  
#include <string>  
#include <vector>  
  
int main()  
{  
using namespace std;  
  
array<std::byte, 200'000> buf;  
pmr::monotonic\_buffer\_resource pool{buf.data(), buf.size()};  
  
pmr::vector<pmr::string> v{&pool};  
for (int i=0; i != 1000; ++i)  
v.emplace\_back("yet another longer string");  
}

When using C++ allocators that are not derived from std::pmr::memory\_resource or are not part of an allocator provided by a library, care is needed to avoid vulnerabilities due to memory leaks due to issues in allocator code.

The library functions std::construct\_at() and std::destroy\_at() are simpler than and preferrable to *placement new* and destructor calls.

STOPPED HERE.

ISSUES WITH SHARED\_PTR AND weak ptr or std::make\_shared(). Using std::shared\_ptr() can cause memory leaks if it is used to create a cyclic data structure that is not internally connected using weak pointers. Y\_shr -> A -> B -> C -> A X\_shr -> A shared\_ptr differentiates between ownership and reference to A Shared\_ptr is a smart pointer that retains shared ownership f an object through a pointer. Shared\_ptr can also own zero objects. A weak\_ptr is a smart pointer that holds a non-owning reference to an object that is managed by shared\_ptr. A weak\_ptr must be converted to a shared\_ptr in order to access the referenced object.

### 6.39.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects, C++ software developers can:

* Apply the avoidance mechanism of ISO/IEC 24772-1:2024 6.39.5
* Refrain from using the C functions malloc(), calloc(), realloc() and free() wherever possible.
* Avoid mixing C-based functions and C++ memory management functions.
* Prefer PMR
* Prefer containers like those in the standard library over any other form of memory management.
* Use smart pointers and their factory functions to allocate and manage heap memory.
* For heap fragmentation issues, use special memory resource objects with appropriate allocation strategies.
* If using std::shared\_ptr in potentially cyclic data structures, break cycles using std::weak\_ptr.
* Use static analysis to prevent uses of C-library memory management functions and direct calls to operators new and delete.
* Use dynamic analysis to detect memory leaks and issues with heap fragmentation.

## 6.40 Templates and Generics [SYM]

### 6.40.1 Applicability to language

The vulnerability as described in ISO/IEC TR 24772-1 clause 6.40 exists in C++. C++ provides the facility *Templates* to support the generic programming methodology.

C++ provides templates for functions, classes(types), and variables (constants). In addition one can form alias templates for class templates. Template parameters can be types, values (including addresses of global variables), and templates. C++ Templates can have variadic template parameters, that mean any number of arguments of a given kind can be used. Concepts are templates that describe constraints on template arguments and can be used to define template parameters or other deduced contexts.

At compile-time, templates are instantiated with concrete template arguments. Function templates as well as class template constructors can deduce the concrete template argument from the types of the function arguments used in a call. For class templates in addition to the implicit deduction guides provided by its constructors, explicit deduction guides can be specified. This mechanism of template-argument deduction allows one to use templates without explicitly mentioning a template argument for each template parameter. For class templates, only those member functions get instantiated that are actually used. Each template instantiation is checked for syntax, concept and type errors.

Each instantiation of a template is compiled separately, which can cause different instantiations from the same original source code to call different actual functions because of overload resolution.

When comes the time for instantiation of a template, there can be different speciations that match the template arguments. There is an ranking of the different (partial) speecializations that the compiler uses for selection. In case of ambiguities, the compiler will fail. This ranking can be influenced by *Concepts* and *SFINAE* (Substitution Failure Is Not An Error). If the chosen specialization compiles but behaves differently than expected, this can be a source of programmer confusion.

**To Be Continued 2 Oct 2023** Templates add another level of complexity to overload resolution.

In case of a function overload set that includes function templates, overload resolution happens before template specialization. This means, any desired behaviour through explicit function template specialization is not considered during overload resolution, only the primary template is used there.

Class template and variable template specializations can provide specific code for a given set of template arguments. Such specializations must be defined in the namespace of the primary template. To prevent confusion and different compilation of identical looking template instantiations, a specialization should either be defined in the same file as the generic template, or in case of a specialization for a specific template argument type, in the file of the definition of that type.

Functions and lambdas that define parameters with the use of auto are implicitly templates without using the template keyword.

Variables defined with the use of auto keyword get their concrete type deduced from their initializer, as if they were function template parameters.

A constructor template or assignment operator template is never a copy or move operation and hence does not prevent the implicit definition of a copy or move operations even if it looks similar.

Due to the two phase compilation model of templates, name lookup can be surprising in class templates with dependent base classes. A name used in the derived class that is defined in the base might be found in an outer namespace instead.

double foo{0};  
template <typename T>   
struct base {  
 int foo;  
};  
template <typename T>   
struct d : base<T>{  
auto bar() {  
return foo; // matches global foo not base<T>::foo [1]  
}  
};

In the above example line [1], in place of foo, either this->foo or the fully qualified name d::foo would refer to the member of the base class.

When used appropriately, templates are suitable for embedded and safety critical systems;

* They provide type-safe generics in contrast to legacy void \*-based or macro-based genericity;
* They have no runtime overhead;
* They do not have any implicit run-time memory usage; and
* They do not generate code for unused operations.

While using template greatly increases type safety, there can be requirements on template arguments that can neither be specified by concepts nor checked by a compiler. For example, sorting elements requires the comparison function to provide a *strict weak ordering* which is a property of the values of the type to be sorted by which are impossible to check at compile time for all possible value combinations.

C++ provides means to restrict template arguments. One is to use concepts, that can prevent instantiating a template, but allow for substituting it with an alternative. A second means is to use static\_assert in a template’s definition to prevent certain instantiations.

template<typename T>  
struct wrapper {  
T x;  
static\_assert(not (std::is\_pointer\_v<T> || std::is\_reference\_v<T>));  
};  
template<typename T>  
wrapper(T )->wrapper<T>;  
  
wrapper<int> w{42};  
wrapper x{&w}; // compile error due to static\_assert  
wrapper<int&>{}; // compile error due to static\_assert

The generic nature of templates require a more elaborate approach to unit tests. Such tests should provide instantiations of the base template and all provided explicit template specializations to ensure that each code path is actually tested. Tests for non-compilability of suppressed instantiations, i.e., through concepts or static\_assert, are also beneficial.

Templates allow to reduce the amount of boilerplate code to write, e.g., by providing consistent definitions of operators. However, defining operator function templates in namespace scope can greatly influence compile times due to potential participation in the overload set, whenever the operator is used in code. In addition such generic operator functions might be picked up in inappropriate places causing programmer confusion. Implementing them as hidden friends in a CRTP base class instead makes using operator function templates feasible (see [6.20 Identifier Name Reuse [YOW]](#YOW)).

template <typename T>  
struct Plus {  
friend constexpr auto operator+(T l, T const &r) {  
 return l += r;  
}  
};  
struct Int: Plus<Int> {  
constexpr auto operator+=(Int const &r) {  
 val += r.val;  
 return \*this;  
}  
constexpr Int(int v):val{v}{}  
int val;  
};  
struct Short: Plus<Short> {  
constexpr auto operator+=(Short const &r) {  
 val += r.val;  
 return \*this;  
}  
constexpr Short(short v):val{v}{}  
short val;  
};  
auto x = Int{4} + Int{38};  
auto y = Short{4} + Short{2};

<!--  
  
(\*We may wish to summarize)\*  
-->

### 6.40.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects, C++ software developers can:

* Follow the avoidance mechanisms of ISO/IEC 24772-1 subclause 6.40.5. and the guidance provided in the different related sections of this document.
* Use static\_assert to prevent the use of inappropriate template arguments.
* Consider using *concepts* to constrain template parameters.
* Prefer *concepts* over *SFINAE*.
* Be aware that a constructor template or assignment operator function template will not replace compiler-provided special member functions.
* For generic operator functions, consider providing them as hidden friends through mix-in class templates.
* Use *qualified-id* or this-> to refer to names that may be found in a dependent base class.
* For template specialization, ensure that specializations are declared as follows:
  1. In the same file as the primary template; or
  2. In the same file as the user-defined type for which the specialization is declared.
* Prohibit specializing function templates.

## 6.41 Inheritance [RIP]

## 6.41.1 Applicability to language

The vulnerability as described in ISO/IEC 24772-1:2024 6.41 is applicable to C++.

In C++, inheritance is defined differently than in most other languages that support inheritance, e.g.: - There is no implicit common base class. - Defining a member with a name that is defined in a base class causes hiding the base class member in the derived class including all overloads with that name. (Full qualification via the base class name is required to access these hidden members.) - Dynamic polymorphism requires the use of references or pointers. - Private virtual member functions can be overridden in derived classes.

A vulnerability similar to namespace issues (see [BJL]) can arise if a new *final overrider* is defined in one of multiple parents, which can result in a silent change to the semantics of an existing program, as shown in the following example.

struct A {  
 virtual void foo ();  
};  
  
struct B : virtual A { };  
  
struct C : virtual A {  
 void foo () override; // late addition  
};  
  
struct D : B, C   
{  
 void bar () {  
 foo (); // calls the late addition  
 }  
};

The implementation of virtual member functions does not take access into account. This can lead to a potentially surprising situation where a virtual function overrides an otherwise inaccessible base class function. This is quite different than the situation for non-virtual private member functions, which cannot be accessed from outside the declaring class or any friends.

Inheriting from a non-polymorphic base class can be used to implement *adaptor* types, i.e., by limiting or extending the base’s functionality. Using private inheritance can prevent treating the derived type object as a base type object. Multiple inheritance adds an additional dimension since a class can inherit the same base class A indirectly via different direct base classes and can result in multiple base class sub-objects of type A, which can result in ambiguous code.

If all classes in a multiple inheritance hierarchy that directly inherit from A use the keyword virtual when inheriting from A, there will be only one base class sub-object of type A in the most derived object. Inconsistently inheriting from A with and without virtual is confusing, because multiple base objects of type A exist. Even when an ambiguity does not currently exist, refering to a base class member using its qualified name reduces the chance that a future modification silently changes the program’s behaviour through accessing a different member with the same name.

In C++, a name declared in a virtual base and inherited multiple times but hidden on any virtual path is considered to be hidden on all virtual paths. Consequently, a hiding declaration introduced later will silently alter the semantics of code refering to this name. The vulnerability does not arise if a multiply-inherited name is found on at least one non-virtual inheritance path, since the use of the name is diagnosed as ambiguous.

C++ favours value-semantics for copy and move operations as well as destruction that conflicts with object-oriented polymorphic behaviour.

* **Virtual Destructor**: Base classes that define virtual member functions will need to also define a virtual destructor and in addition need to care about the copy and move operations, otherwise deleting a dynamically-allocated derived object via a base class pointer will cause undefined behaviour [[EWF]](#EWF).
* **Slicing**: A common failure is to not eliminate implicitly callable copy and move operations in base classes that can lead to accidental copying of a base class sub-object via a base class reference that actually refers to a derived object. Preventing implicit copy and move operations in base classes defining virtual member functions is a common mitigation.
* **Incomplete Copy**: When a derived class defines its own non-defaulted, non-deleted copy or move operations, care must be taken to actually copy and move all base class sub-objects as well. Omitting a base class when defining copy or move constructors means the default construction of a base class object happens. Not invoking a base class assignment in the definition of copy and move assignment operators will cause the base class retaining its previous members and not obtaining the source object’s base members. None of these omissions are a compile error and none are an issue for empty bases.

If a “using-declaration” refers to a constructor of a direct base class, all constructors of that base are candidates for the initialization of the derived class. If a base class constructor is selected to initialize the derived type, any other bases or members of the derived type will be initialized as if by an = default default constructor. This can leave some members uninitialzed, that can result in undefined behaviour.

If a using-declaration refers to a protected member of a base class, its accessibility can be changed, which destroys the encapsulation of the member.

If a base class overloads operator new and operator delete, any derived classes will inherit those operations. If the base class’ operator new and/or operator delete assume the size of the objects being allocated are all the size of the base class and they are not all the same size, then this can result in undefined behaviour.

The mechanisms of failure from ISO/IEC 24772-1:2024 clause 6.41 can be mitigated in C++ as follows:

* *Execution of malicious redefinitions* can be prevented by use of final on each member function to generate compiler diagnostics when overriding is not permitted.
* *Accidental redefinition* can be mitigated by a project mandate to use the override or final special identifiers when overriding a virtual member functions.
* *Accidental failure of redefinition* can be prevented by using override on each member function intended to be redefined to generate compiler diagnostics when overriding does not apply.
* *Breaking of class invariants* can be avoided by proper initialization even with the default constructor and by defining data members private to ensure that the functions with the appropriate functionality are called and thus class invariants are preserved. If copy and move operations are user-defined in a derived class they must ensure to call the corresponding base class operations.

## 6.41.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects, C++ software developers can:

* Apply the avoidance mechanisms of ISO/IEC 24772-1 clause 41.5.
* Avoid multiple inheritance.
* Ensure that shallow-copying or shallow-move of inheritance hierarchies cannot occur, e.g. by defining the appropriate member operation =delete (see clause 6.38 Deep vs. Shallow Copying [YAN]).
* If the implementation of copy-operations or move-operations in a derived class is mandatory, then statically ensure that all calls are to the corresponding base classes’ operations.
* Prefer composition over inheritance.
* Keep inheritance hierarchies shallow.
* Avoid the use of virtual member functions unless unbounded run-time polymorphism can be justified.
* If a class has any virtual functions, ensure it has a virtual destructor.
* Use the special identifier override whenever overriding a virtual member function.
* Use the special identifier final to prevent an unexpected overriding of that function deeper in the hierarchy.
* Require the use of final or override when overriding a member function.
* Do not assume that a private virtual function cannot be overridden.
* Consider using fully-qualified names to address members of a base class.
* When defining an overload that hides overloads in a base class, consider adding a *using declaration* for the name from the base class.
* Avoid the use of using-declarations that refer to constructors in types where default initialization is not appropriate for base classes of the derived type or members of the derived type.
* Avoid using a “using-declaration” to promote the accessability of a base class member.
* Prohibit mixing virtual and non-virtual inheritance of the same base class in a hierarchy.
* Preserve class invariants and declare all class members as protected or private whenever appropriate.
* Use static analysis tools, including compilers from different sources, to:
  + detect hiding of member names in virtual base classes, and
  + identify methods that are accidentaly turned into overrides when equally-named methods are added to parents.

## 6.42 Violations of the Liskov Substitution Principle or the Contract Model [BLP]

### 6.42.1 Applicability to language

The vulnerability as documented in ISO/IEC 24772-1 clause 6.42 applies to C++. C++ leaves verification of the correctness of an overridden call to the programmer.

The vulnerability can be mitigated by a style of programming that uses wrapper functions to check preconditions, calls a virtual function to perform the required functionality and subsequently checks the postconditions before returning.

An example is provided below.

class Base {  
 private:  
 virtual int function\_to\_override( int x ) = 0;  
 // ...  
  
 public:  
 int interface\_to\_overridden\_function( int x ) {  
 check\_preconditions( x );  
 const auto saved = data\_saved\_for\_postcondition( x );  
 auto result = function\_to\_override( x );  
 check\_postconditions( x, saved, result );  
 return result;  
 }  
 // ...   
 };

A future version of the C++ standard is expected to contain “contracts”, which formalize the notion of preconditions, postconditions and contract assertions.

### 6.42.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects, C++ software developers can:

* Apply the avoidance mechanisms of ISO/IEC 24772-1 6.42.5
* Use static analysis tools that identify violations of preconditions and postconditions.
* Ensure that all invariants of a derived class are preserved by all public operations on its public base classes.

See also C++ Core Guidelines C.120, C.121, C.122, C.126, C.127, and C.129 through C.133.

## 6.43 Redispatching [PPH]

### 6.43.1 Applicability to language

The vulnerability as described in ISO/IEC TR 24772-1:2019 clause 6.43 exists in C++ for virtual functions, except for constructors and destructors which are not dispatching. An example of the infinite recursion is:

#include <iostream>  
  
class A {  
public:  
 virtual void f() { std::cout << "A::f()\n"; }  
 virtual void g() { std::cout << "A::g()\n"; A::f(); } //call to f() will not dispatch.  
 virtual void h() { std::cout << "A::h()\n"; }  
 virtual void i() { std::cout << "A::i()\n"; h(); } //call to h() will dispatch  
 //showing the vulnerability  
};  
  
class B : public A {  
public:  
 void f() override { std::cout << "B::f()\n"; g(); }  
 void h() override { std::cout << "B::h()\n"; i(); }  
};  
  
int main() {  
 B b;  
 A \* pA = &b;  
 pA->f(); // no problem  
 std::cout << "---\n";  
 pA->h(); // infinite recursion  
}

In C++, the call to a member function can be qualified, as shown in the above example, and avoids the vulnerability.

### 6.43.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects, C++ software developers can:

* At a call site, consider whether virtual dispatch is desired: if not, construct the call using the qualified name.
* Be suspicious of any call from a virtual member function of the derived class to any member function of any of its base classes.

## 6.44 Polymorphic variables [BKK]

### 6.44.1 Applicability to language

This vulnerability as described in ISO/IEC TR 24772-1:2019 applies to C++. In addition to the upcast and downcast issues addressed in that document, this clause also addresses cross-casting, which is unique to C++. For further type system related issues see subclause [Type System[IHN]](#IHN).

C++ provides language mitigations to help avoid the problems as follows:

Since C++ supports multiple inheritance, up-casting, down-casting, and cross-casting operations can be used to switch to different (pointer/reference) types in the inheritance hierarchy of a specific object, i.e.,

* *up-casting* is casting an object to an ancestor type in the object's type inheritance hierarchy.
* *down-casting* is casting an object to a descendent type in the object's type inheritance hierarchy, and,
* *cross-casting* is casting an object to a sibling/cousin (possibly removed) type in the object's type inheritance hierarchy with multiple inheritance.
* Unsafe casts, which include C-style casts and reinterpret\_cast, can cast to unrelated arbitrarily structured types. This allows reading and modifying arbitrary memory areas. See subclause [[6.11 Pointer Casting and Pointer Type Changes] [HFC](#HFC) for more details.

Developers should be aware that virtual member functions can be overridden in derived classes, even if they are private.

Given the following:

struct Z { int z; virtual ~Z() { } };  
struct Y { int y; virtual ~Y() { } };  
struct A : Z { int a; };  
struct B : virtual A { int b; };  
struct C : virtual A, Y { int c; };  
struct D : B, C { int d; };  
D d\_inst;

then these examples demonstrate upcasts, downcasts, and crosscasts:

**Upcasts:**

B\* b\_ptr = &d\_inst; // implicit  
C& c\_ref = d\_inst; // implicit  
Z\* z\_ptr = static\_cast<Z\*>(&d\_inst);  
Y\* y\_ptr = dynamic\_cast<Y\*>(&d\_inst);

**Downcasts:**

D& d\_ref = dynamic\_cast<D&>(\*y\_ptr);  
D\* d\_ptr = static\_cast<D\*>(b\_ptr);

**Crosscasts:**

C\* c\_ptr = dynamic\_cast<C\*>(b\_ptr);  
Y\* y\_ptr2 = dynamic\_cast<Y\*>(b\_ptr);  
C\* c\_ptr = static\_cast<C\*> (static\_cast<D\*>(b\_ptr));

and notes the following about such:

Upcasts:

* are the only ones that can be performed implicitly
* can cause object slicing when a copy of a base class object is created from a derived class object.
* can also be done with dynamic\_cast or static\_cast

Downcasts

* are explicit;
* can be done safely with dynamic\_cast
* dynamic\_cast requires appropriate portions of inheritance to be polymorphic (i.e. has virtual members);
* can be done using static\_cast which is unchecked and may be unsafe;

Crosscasts:

* are explicit
* can be done safely with a single call to dynamic\_cast which requires appropriate portions of inheritance to be polymorphic (i.e. has virtual members).
* can often be done with a chain of static\_casts traversing the inheritance hierarchy, which is almost always unsafe.

Deleting derived objects via a base class pointer is undefined behavior [[EWF]](#EWF), unless the base class declares a virtual destructor.

### 6.44.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects, C++ software developers can:

* Follow the avoidance mechanisms of ISO/IEC 24772-1 clause 6.44.5.
* In a base class declaring virtual member functions, define a virtual defaulted destructor (Core Guidelines C.35) and make the class non-copyable (Core Guidelines C.130).
* Avoid designs relying on downcasts or crosscasts; rely on proper virtual member functions in the base classes instead.
* Avoid explicit upcast, rely on implicit conversion.
* If a downcast or a crosscast is needed, prefer using dynamic\_cast since it is checked.
* Ensure that all invariants of a derived class are preserved by all public operations on its public base classes. If this cannot be ensured, make the base class private, or avoid inheritance.
* Avoid attempting to navigate class hierarchies using C-style casts or reinterpret\_cast.
* Prohibit deletion of a polymorphic object without a virtual destructor; including using std::unique\_ptr<base> from a unique pointer allocated with std::make\_unique<derived>()

See also C++ Core Guidelines ES.48, ES.49, C.146, C.147, C.148 and C.153. source: OOP52-CPP?

## 6.45 Extra Intrinsics [LRM]

### 6.45.1 Applicability to the language

The vulnerability as described in ISO/IEC 24772-1 clause 6.45 applies to C++ as explained below.

C++ implementations are allowed to provide built-in functionality but are restricted to a specific naming schema reserved by the standard. For example, names containing a double underscore or that begin with an underscore and a capital letter are reserved for that purpose. See ISO/IEC 14882 clause [Lex.name]. The use of such names by the programmer is forbidden by the language. Language processors are not required to prohibit such usage, hence the vulnerability exists.

The standard restricts definitions in reserved namespaces, such as std (see ISO/IEC 14882 clause [namespace.constraints]). In addition, specializing a template from namespace std is restricted (see ISO/IEC 14882 clause [namespace.std]) unless explicitly allowed, for example, see ISO/IEC 14882 clause [unord.hash].

### 6.45.2 Avoidance mechanisms for program users

To avoid the vulnerability or mitigate its ill effects, C++ software developers can:

* Prohibit identifiers containing a double underscore \_\_.
* Avoid identifiers starting with \_, except when used for a literal suffix.
* Ensure user-defined literal suffixes start with a single \_.
* Prohibit the declaration of names in reserved namespaces.
* Prohibit the specialization of a restricted template.
* Use compiler warnings and static analysis tools to detect and report violations of the above.

## 6.46 Argument Passing to Library Functions [TRJ]

### 6.46.1 Applicability to language

The vulnerability as described in ISO/IEC TR 24772-1:2019 clause 6.46 is applicable to C++.

Libraries that supply objects or functions are in most cases not required to check the validity of parameters passed to them. In those cases where parameter validation is required there might not be adequate parameter validation.

When calling a library, either the calling function or the library may make assumptions about parameters. For example, it may be assumed by a library that a parameter is non-zero so division by that parameter is performed without checking the value. Sometimes some validation is performed by the calling function, but the library may use the parameters in ways that were unanticipated by the calling function resulting in a potential vulnerability. Even when libraries do validate parameters, their response to an invalid parameter is usually undefined and can cause unanticipated results.

This vulnerability applies in particular to C++ libraries which are designed for high efficiency; responsibility for satisfying the preconditions for most functions rests with the caller. When these preconditions are not met, the result will be undefined behaviour [[EWF]](#EWF). In addition, error conditions are specified by the language for specific functions, such as raising an exception, returning an error code or a known value, such as NaN.

### 6.46.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects, C++ software developers can:

* Follow the avoidance mechanisms of ISO/IEC 24772-1 clause 6.47.5.
* Use translation modes provided by the implementation to perform addition analysis or checking, such as contracts checks, or instrumentation of executing code.
* Pay attention to the distinction between precondition violation and error conditions in library documentation: the former results in undefined behaviour; the latter results in defined but possibly unwanted behaviour.

## 6.47 Inter-language Calling [DJS]

### 6.47.1 Applicability to language

The vulnerability as described in ISO/IEC TR 24772-1:2019 clause 6.47 is applicable to C++.

C++ is a multi-paradigm language with a number of features that do not interface simply with other language systems. It is left to the implementation team the task of converting the results of these paradigms to constructs that can cross an interface for further processing in other languages.

C++ compilers provide an application binary interface (ABI) that delineates areas of interoperability with other languages or other C++ compiler/runtime systems. An ABI includes calling conventions, data layout, error and exception handling and return conventions, name mangling, data model, initialization of memory, and linkage to operating systems and libraries.

C++ compilers implement a C++ language linkage and a C language linkage. It is implementation-defined what other languages the implementation supports. Alternatively, other language systems provide linkages to C systems[^3](Ada has developed a standard for interfacing with C. Fortran has included a Clause 15 that explains how to call C functions.), leaving the developer the task of channeling everything through this common language system.

### 6.47.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects, C++ software developers can:

* Follow the avoidance mechanisms of ISO/IEC 24772-1 clause 6.47.5
* Use standard layout types for the interoperable interfaces.
  + Use language linkage facilities that support the languages being
  + used.
  + EXP56-CPP. Do not call a function with a mismatched language
  + linkage (-> 6.47)
  + EXP60-CPP. Do not pass a nonstandard-layout type object across
  + (-> 6.47 ?)  
    execution boundaries
* Be aware that the static initialization phase and dynamic initialization for every language system are required before the system begins execution
* Be aware that C++ exceptions are not usually compatible with exceptions in other languages.
* Segregate outgoing cross-language interfacing code into functions that present a C++ interface to the C++ code and implements that interface by calling code compatible with the other language system. Similarly implement incoming cross-language interfaces by providing simplified functions that presents a simplified (C or other language) interface and is implemented by calling C++ code with the correct style.
* Separate the interfacing code from the code containing the main functionality

See also the C++ Core Guidelines CPL.3.

*AI 63-6 – group – add the guidance from 6.47.2 Interoperability into the Core Guidelines.*

## 6.48 Dynamically-linked Code and Self-modifying Code [NYY]

### 6.48.1 Applicability to language

The vulnerability described in ISO/IEC 24772-1:2024 applies to C++. As for the C programming language, the process of linking and loading is outside the scope of the C++ standard.

In C++, the conversion of a function pointer to another type (to misdirect a pointer, for example) requires the use of a C-style cast or reinterpret\_cast. Such casts can result in unspecified behaviour, undefined behaviour or implementation-defined behaviour.

### 6.48.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects, C++ software developers can:

* Follow the avoidance mechanisms of ISO/IEC 24772-1 clause 6.48.5 and ISO/IEC 24772-3 clause 6.48.2.
* Follow the avoidance mechanisms of 6.11.2 Pointer type conversions [HFC].
* Document and carefully review all uses of reinterpret\_cast or C-style casts.

## 6.49 Library Signature [NSQ]

### 6.49.1 Applicability to language

The vulnerability as enumerated in ISO/IEC 24772-1 clause 6.49 applies to C++.

As a mitigation, the C++ extern "C" linkage specifier usually provides simple interoperability with libraries using the C application binary interface (ABI).

### 6.49.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects, C++ software developers can:

* Follow the avoidance mechanisms of ISO/IEC 24772-1 clause 6.49.5.
* Follow the avoidance mechanisms of clause [6.47 Inter-language Calling [DJS]](#DJS) as applicable.

## 6.50. Unanticipated Exceptions from Library Routines [HJW]

### 6.50.1 Applicability to language

The vulnerability as documented in ISO/IEC TR 24772-1:2019 clause 6.50 exists for C++. In particular the issue of the failing dynamic initialization of namespace-scope objects exists in C++.

When dynamic initialization of a namespace-scope object fails with an exception, the exception cannot be caught and the program is terminated. Function-scope static objects, in contrast, are initialized the first time execution passes through the declaration. Using function-scope static objects in preference to dynamic initialization ensures that there is always an enclosing function that could catch the exception.

exception\_prone\_type troubling\_object;  
// An exception from the constructor could cause termination.  
// The following function always returns a reference to the same object,  
// which is initialized the first time this function is called.  
  
// If initialization fails, it will be retried on the next call.  
exception\_prone\_type& safer\_object()  
 {  
 static exception\_prone\_type the\_safer\_object;  
 return the\_safer\_object;  
 }

### 6.50.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects, C++ software developers can:

* Expect functions not marked noexcept to throw exceptions of arbitrary type, excluding all destructors, which are are implicitly noexcept.
* Follow the avoidance mechanisms of clause 6.36.2 for catching and handling exceptions.
* Prefer function-scope static objects to namespace-scope objects for objects needing dynamic initialization.

## 6.51 Pre-processor Directives [NMP]

### 6.51.1 Applicability to language

The vulnerability as described in ISO/IEC TR 24772-1:2019 clause 6.51 applies to C++.

The C++ pre-processor allows the use of macros that are text-replaced before compilation.

Function-like macros look similar to functions but have different semantics. Because the arguments are text-replaced, expressions passed to a function-like macro may be evaluated multiple times. This can result in unintended and undefined behaviour [[EWF]](#EWF) if the arguments have side effects or are pre-processor directives. Additionally, the arguments and body of function-like macros should be fully parenthesized to avoid unintended and undefined behaviour.

The following code example demonstrates undefined behaviour when a function-like macro is called with arguments that have side-effects (in this case, the increment operator) .

#define CUBE(X) ((X) \\* (X) \\* (X))  
// ...  
 int i = 2;  
 int a = 81 / CUBE(++i);

The above example could expand to:

int a = 81 / ((++i) \* (++i) \* (++i));

which has undefined behaviour so this macro expansion is difficult to predict.

Another mechanism of failure can occur when the arguments within the body of a function-like macro are not fully parenthesized. The following example shows the CUBE macro without parenthesized arguments.

#define CUBE(X) (X \\* X \\* X)  
// ...  
int a = CUBE(2 + 1);

This example expands to:

int a = (2 + 1 \* 2 + 1 \* 2 + 1)

which evaluates to 7 instead of the intended 27.

### 6.51.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects, C++ software developers can:

* Replace function-like macros with constexpr inline functions where possible.
* Replace normal macros with constexpr variables where possible.
* Replace conditional compilation with the preprocessor with if constexpr where possible, e.g., in function bodies, including cases where compile-time define of a macro (as empty) controls if a macro definition is used to expand to an empty statement or another statement.
* Replace preprocessor include directives with module import where possible.
* Prefer std::source\_location mechanisms over employing macros that use \_\_LINE\_\_, \_\_FILE\_\_, or \_\_func\_\_.
* If a function-like macro must be used, ensure that its parameters and body are parenthesized.
* In a function-like macro, ensure that each argument is evaluated at most once.
* Prohibit embedding the following in a function-like macro: pre-processor directives; or side-effects such as an assignment, increment/decrement, volatile access, or function call.
* Only use macros for *include guards*, to control conditional compilation, or when the macro’s definition requires token pasting (##.{.cpp}) or stringification (#) of macro arguments.
* Prohibit the use of macro paramters that are used as argument for ## or # in the macro.

## 6.52 Suppression of Language-defined Run-time Checking [MXB]

### 6.52.1 Applicability to language

With the exception of the macro assert, the vulnerability as described in ISO/IEC TR 24772-1:2019 clause 6.52 does not apply to C++, because there is no language-defined runtime checking. Macro assert is defined by the standard but is invoked by the programmer, hence is not a language-defined check.

C++ libraries, however, often provide run-time checks which meet the criteria of this vulnerability. Also, compilers and other tools commonly provide means to perform such runtime checks.

### 6.51.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects, C++ software developers can use the avoidance mechanisms of ISO/IEC 24772-1 clause 6.52.5 with respect to library and compiler-provided checks, which will almost always require the explicit enabling the checks.

## 6.53 Provision of Inherently Unsafe Operations [SKL]

### 6.53.1 Applicability to language

The vulnerability as described in ISO/IEC 24772-1:2019 clause 6.53 applies to C++. In particular, anything described by ISO/IEC 14882:2023 as “undefined behaviour” [[EWF]](#EWF) is unsafe.

### 6.53.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects, C++ software developers can:

* Use the avoidance mechanisms of ISO/IEC 24772-1:2024 and ISO/IEC TR 24772-3:2020 clause 6.53.5.
* Enable checks that warn about unsafe operations.
* Use static analysis tools to detect unsafe constructs.

## 6.54 Obscure Language Features [BRS]

### 6.54.1 Applicability of language

The vulnerability as described in ISO/IEC 24772-1 clause 6.54 applies to C++.

C++ is a rich language and provides facilities for a wide range of application areas with a long history of evolution. The latter means that evolution of the language also means an evolution of best and safe practices. This means that code can look obscure, because it either uses obsolete or very modern language idioms.

### 6.54.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects, C++ software developers can:

* Follow the avoidance mechanisms of ISO/IEC 24772-1 clause 6.54.5.
* Use static analysis tools or multiple compilers to help identify occurrences of obscure language features.

## 6.55 Unspecified Behaviour [BQF]

### 6.55.1 Applicability of language

The vulnerability as described in ISO/IEC 24772-1:2019 clause 6.55 applies to C++.

In the following example, the value of size can be different than the length of data because the extent of data must be calculated at compile time but the calculation of size can be determined at compile time or executed at runtime, which is one source of unspecified behaviour.

char data[1 + int(1 + 0.2 - 0.1 - 0.1)] = { }; // compile-time evaluation  
int size = 1 + int(1 + 0.2 - 0.1 - 0.1); // compile-time or run-time evaluation  
char f() { return data[size-1];} // Possible buffer overflow

### 6.55.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects, C++ software developers can:

* Follow the avoidance mechanisms of ISO/IEC 24772-1 clause 6.55.5.
* Use static analysis tools and multiple compilers/tools to help identify occurrences of unspecified behaviour.

## 6.56 Undefined Behaviour [EWF]

### 6.56.1 Applicability to language

The vulnerability as described in ISO/IEC 24772-1 clause 6.56 applies to C++. In ISO/IEC 14882:2023, the terms “undefined behaviour” and “ill-formed, no diagnostic required” expose situations to be avoided.

For the special case of constant expressions, compilers are required to detect and reject undefined behaviour during the evaluation of constant expressions in the core language. It is unspecified whether or not undefined behaviour in the standard library is detected during constant expression evaluation.

C++ compilers and static analysis tools can often detect code that can lead to undefined behaviour and provide diagnostic messages to that effect. Use of multiple compiler tool chains during development increases the likelyhood that such behaviours will be detected. Similarly, runtime tools such as address sanitizers and thread sanitizers can be used during development to detect and report concrete instances of undefined behaviour. In addition, recent C++ language revisions have provided safer alternatives some standard library calls that can lead to undefined behaviours. For example,the calls std::stoi or std::from\_chars provide equivalent functionality to std::atoi but without the the risk of undefined behaviour.

### 6.56.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects, C++ software developers can:

* Follow the avoidance mechanisms of ISO/IEC 24772-1 clause 6.56.5.
* Exploit compile-time evaluation whenever possible to detect undefined behaviour.
* Use static analysis tools in addition to the compiler to help identify occurrences of undefined behaviour.
* Augment static analysis tool usage with runtime tools such as ASAN (address sanitizer) and related tools.
* Use multiple compilers/tools and different optimization levels to increase the chance of identifying constructs that have undefined behaviours.
* Where the C++ language or library provide both defined behaviour mechanisms and undefined behaviour mechanisms, mandate the use the mechanisms with defined behaviours.

## 6.57 Implementation–defined Behaviour [FAB]

### 6.57.1 Applicability to language

The vulnerability as described in ISO/IEC 24772-1:2019 clause 6.57 applies to C++. In ISO/IEC 14882, the term “implementation-defined” is used to describe implementation-defined behaviour. In addition, the C++ standard provides a dedicated index titled, “Index of implementation-defined behavior”.

In deciding whether or not to use a feature with implementation-defined behaviours, it is important be cognizant that:

* A given implementation may change or remove an implementation-defined behaviour; and
* Programs that are specifically intended for a particular implementation may in the future be ported to another environment or sections reused for future implementations.

In either situation, the implementation-defined behaviour will become problematic.

### 6.57.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects, C++ software developers can:

* Use the avoidance mechanisms of ISO/IEC 24772-1 clause 6.57.5.
* Eliminate to the extent possible any reliance on implementation-defined behaviour from programs in order to increase portability.
* Use multiple compilers/tools to increase your chance of identifying constructs that have implementation-defined behaviours.
* Refer to “Index of implementation-defined behavior” for the definitive list of C++ implementation-defined behaviours and understand the relevant compiler documentation for each of these behaviours.

## 6.58 Deprecated Language Features [MEM]

### 6.58.1 Applicability to language

The vulnerability as described in ISO/IEC 24772-1 clause 6.58 applies to C++. Appendix D “Compatibility features” of ISO/IEC 14882:2020 enumerates the deprecated features. The C++ attribute [[deprecated]] allows library writers and users to mark deprecated declarations.

Although backward compatibility is sometimes offered as an option for compilers so one can avoid changes to code to be compliant with current language specifications, updating the legacy software to the current standard is a better option.

### 6.58.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects, C++ software developers can:

* Follow the avoidance mechanisms of ISO/IEC 24772-1 clause 6.58.5.
* Enable compiler options or use static analysis tools that identify the use of deprecated features.
* Apply the [[deprecated ("\*reason\*")]] attribute to obsolete declarations that exist only for backward compatibility.

## 6.59 Concurrency – Activation [CGA]

### 6.59.1 Applicability to language

The vulnerabilities as specified in ISO/IEC 24772-1 clause 6.59 do not apply to C++, as long as the standard library facilities for creating threads are used. If the C standard library mechanisms or threading libraries from POSIX or other applications are used to create threads, then the vulnerabilities apply as described in ISO/IEC 24772-3 clause 6.59. This subclause will examine the C++ standard mechanisms.

Creating a thread using the std::async function or the std::thread or std::jthread results in the constructor synchronized with the thread creation site. Unless the thread is detached, join() must be called when using std::thread or std::terminate will be invoked upon destruction of the thread object. This can be avoided by using std::jthread instead.

Failure to create or start a thread due to lack of system resources will cause an exception to be thrown to the creating thread thus the thread object never exists. For the vulnerabilities with unhandled exceptions see clause [6.36 Ignored error status and unhandled exceptions [OYB]](#OYB).

C++ provides other ways to create parallel executing functions via - the std::async() call or the std::packaged\_task() functions; - Co-routines (describe and reference clause 7.x on coroutines.

If std::async() threads are created without a launch policy, then system determines whether or not the thread is run lazily (in the current thread) or in a new thread. The launch policy std::launch::async or std::launch::deferred controls whether each asynch thread creation creates new threads or executes the call on demand. When a launch policy is std::launch::async then each call to std::async() will always create a new thread which can lead to resource exhaustion. When a launch policy is std::launch::deferred() the function will execute on the thread that executes the respective get() function for that async function.

C++ provides a mechanism to construct and control the execution of a function on a user-controlled potentially asynchronous thread via the std::packaged\_task mechanism that allows access to the result via a std::future object. Each packaged\_task object can be executed at most once and can be executed on the current thread or on a different thread.

There are a number of vulnerabilities that are possible once the thread is created using std::thread or std::jthread. See [6.63 lock protocol errors [CGM]](#CGM). Each std::thread object must have its .join() member function called before the object is destroyed unless its .detach() member function was previously called. This is not required for threads created as std::jthread. If .detach() was called on thread object then any calls to .join() will result in an exception.

Any exception thrown within a thread’s function needs to be handled by that thread, otherwise such an exception will cause program termination. For handling such termination see clause [6.62 Concurrency - Premature termination [CGS]](#CGS). Employing a std::packaged\_task object as the thread function allows to pass an exception to the holder of its std::future object.

A detached thread will execute until its thread function ends or until the program is terminated. The std::jthread type allows to employ cooperative thread termination through std::stop\_token See [6.60 Concurrency – Directed termination [CGT]](#CGT) or [6.62 Concurrency – Premature termination [CGS]](#CGS).

### 6.59.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects, C++ software developers can:

* When using non-standard thread facilities, e.g., by making use of std::thread::native\_handle, use the avoidance mechanisms of ISO/IEC 24772-1 clause 6.59.5.
* Prefer std::jthread over std::thread for explicit thread creation.
* When handling the exception thrown due to failure to create a thread, ensure that potentially dependent other threads are informed.
* Prefer the use of managed threading constructs such as std::async or C++ parallel algorithms instead of explicit thread creation and management.
* Prefer being explicit about running a function in a deferred way with std::packaged\_task instead of using std::async() without specifying a launch policy.
* Consider using cooperative thread termination through std::stop\_token, especially for detached threads.

## 6.60 Concurrency – Directed termination [CGT]

### 6.60.1 Applicability to language

This vulnerability as specified in ISO/IEC 24772-1 clause 6.60 is mitigated in C++, as long as the standard C++ library facilities for threads are used. For issues associated with threades created using the standard C library, see ISO/IEC 24772-3 clause 6.60.

C++ does not provide the means to terminate a C++ thread asynchronously, except in the case that std::terminate() is called. Instead C++ allows cooperative termination through the use of std::stop\_token, however, a thread instructed by a stop request to cease execution can ignore such a request. For example, using std::jthread::request\_stop() to send a stop request to the started thread, the created thread can have a thread function that never handles such a stop request.

void some\_function(int value);  
void stoppable\_function(std::stop\_token tok, int value);  
  
int main()  
{  
 using namespace std;  
 std::jthread t(some\_function,10); // stop\_token ignored since not provided  
 std::jthread t2(stoppable\_function,100); // stop\_token passed  
 t.request\_stop(); // no-op as ignored in t  
 t2.request\_stop(); // stop\_token `tok`{.cpp} signalled in `t2`{.cpp},  
 // returns `true`{.cpp} showing `stop\_token`{cpp} set  
}  
  
void stoppable\_function(std::stop\_token tok, int value) {  
 using namespace std::literals;  
 while (!tok.stop\_requested()) //  
 {  
 std::osyncstream(std::cout) << value++ << '\n';  
 std::this\_thread::sleep\_for(200ms);  
 }  
}  
  
void some\_function(int value)   
{   
 using namespace std::literals;  
 for (int i =0; i<10; ++i)  
 {  
 std::osyncstream(std::cout) << value++ << '\n';  
 std::this\_thread::sleep\_for(200ms);  
 }  
}  
```{.cpp}  
 bool OK = t2.request\_stop(); }  
  
void stoppable\_function(std::stop\_token tok, int value) {  
 while (!stop\_token.stop\_requested()) //  
 {  
 std::cout << value++ << ' ' << std::flush;  
 std::this\_thread::sleep\_for(200ms);  
 }  
void some\_function(int value) {  
 for (int i = 0; i<10; i++) //  
 {  
 std::cout << value++ << ' ' << std::flush;  
 std::this\_thread::sleep\_for(200ms);  
 }  
  
 std::cout << std::endl;  
}

Note that in the while loop in ’stoppable\_function(){.cpp}, if stop\_token.stop\_requested had not been checked, it would continue to run until the program itself is terminated. In the above example, at the end of the main function the destructors of the thread objectst{.cpp} andt2{.cpp} will callthis->request\_stop(){.cpp} andthis->join(){.cpp}. If one of the thread functions never returns, then the correspondingjoin()`{.cpp} call will block.

Other programmed mechanisms can be constructed to cause another thread to complete, such as setting a shared variable to a known value that the target thread reads and then terminates itself.

A common programming mistake is to create threads with the intention that they will participate in a joint computation, but missing the join() operation in one or more threads. If jthreads are used for the threading, all threads are joined with the thread that created them.

Even if a thread is successfully terminated, the thread that initiated the termination should take action to clean up after the terminating thread,such as releasing memory

Discussion: Do we need to discuss termination of tasks?

## 6.60.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects, C++ software developers can:

* For threads created using the C library, follow the avoidance mechanisms of ISO/IEC TR 24772-3.
* Follow the avoidance mechanisms of ISO/IEC 24772-1 clause 6.60.5.
* Use code reviews and static analysis tools if available, to ensure that threads correctly terminate.

## 6.61 Concurrent Data Access [CGX]

### 6.61.1 Applicability to language

C++ has threading and shared access to variables which have the vulnerabilities described in ISO/IEC 24772-1 clause 6.61.1. Accessing mutable data from multiple threads without synchronization is undefined behaviour [[EWF]](#EWF). C++ provides atomic types, mutual exclusion primitives and higher-level synchronization mechanisms in the standard library to prevent such data races. Sharing non-volatile constant data or not sharing data at all does not incur data races. For example, avoiding pointer-like types for thread function parameters guarantees that there is no sharing of the argument values.

The C++ memory model defines how memory accesses can be reordered. The default memory order guarantees sequential consistency. For increased parallel execution other memory orders can be used but require expert-level of care.

NOTE: The above sentence needs explanation or rewrite.

C++ programs that used multiple threads before the C++11 standard might employ volatile variables to guarantee race free data access. This practice is not sanctioned by current standards, you cannot use volatile for synchronization. The use of volatile std::sig\_atomic\_t only prevents a data race between a signal handler an the execution thread running the signal handler.

The algorithm library allows specifying non-sequential execution policy for most algorithms. Employing parallel (std::par) or vectorized (std::unseq, std::par\_unseq) execution policies may have additional requirements on data layout of the underlying ranges, especially to avoid data races.

Primitives requiring explicit locking and unlocking the pairwise application of these operations is supported implicitly by std::scoped\_lock, even when an exception might be thrown. In addition to scope-based locking, std::scoped\_lock guarantees deadlock-free locking of multiple mutexes.

Variables with thread-local storage duration cannot create data races, as long as their address is not shared with other threads.

Discuss sharing of memory between tasks!! i.e. Don’t do it!?

### 6.61.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects, C++ software developers can:

* Place all mutable data in memory accessible to only one thread at a time;
* Use the appropriate standard library features for synchronization of shared mutable data, in particular use std::scoped\_lock or the other scoped-based locking library facilities;
* Prohibit calling lock and unlock facilities directly.
* Forbid the use of volatile for synchronization;
* Avoid specifying memory orders on atomic operations other than sequential consistency;
* Use atomic variables or fences for synchronization of data shared from a signal handler, employ volatile sig\_atomic\_t only in a single-threaded environment;
* Pass all thread function arguments by non-pointer-like values.

## 6.62 Concurrency – Premature Termination [CGS]

### 6.62.1 Applicability to language

The vulnerability as described in ISO IEC 24772-1 6.62 applies to C++.

For concurrent entities created using the C library, apply the avoidance mechanisms documented in ISO IEC TR 24772-3 clause 6.62.

C++ standard threads

Within C++ standard thread concurrency, the vulnerability as specified in ISO/IEC 24772-1 clause 6.62 can arise as follows: - as part of the termination of the complete program resulting from an explicit call to a program-terminating function in any thread in the program or an implicit call on std::terminate, for example, - as a result of an unhandled exception in any thread function when using thread or jthread - as a result of an unhandled exception in any thread function when using C++ parallel algorithms - as a result of faulty inter-thread asynchronous communications that results in unplanned termination of a communicating thread.

Errors relating to improper shutdown and cleanup are addressed in 6.36 Unhandled errors and exceptions(??) If non-standard C++ concurrency is used, the vulnerability and avoidance mechanisms are as specified in ISO/IEC 24772-1 and ISO/IEC 24772-3.

(see https://en.cppreference.com/w/cpp/error/terminate for more termination conditions)

Joining a thread causes the joining thread to await the joined thread’s termination before continuing. Useful for executing in parallel and then proceeding after the dispatched work is complete, but does not notify the joining task if the termination was premature.

C++ 2020 provides callbacks in the form of stop\_callback to notify the setting thread when a thread of interest has been terminated. It also provides stop\_token for a thread to query it is being instructed to terminate. (should go in 6.60 if not there already).

The terminating thread can pass back whatever is programmed back to the `join’ function, which can include notice of premature termination, for example by returning a pointer to the handled exception that identifies the error that caused premature termination.

The semantics of C++ is that all children of the main program will prematurely terminate if the main program terminates and final results from such threads will not be delivered . It is necessary to join the main program to all its children to ensure that children are not silently terminated prematurely.

Premature termination of C++ Tasks

Tasks can abort or terminate or cancelled (last one - by the scheduler) but these affect the whole program containing the thread. Therefore, the only way for a task to terminate is to receive a terminate request and transfer to the end of its code where the parent that initiated it will be waiting at the future.

### 6.62.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects, C++ software developers can:

* Use the avoidance mechanisms of ISO/IEC 24772-1 clause 6.63.5.
* Use low-level operating system primitives or other APIs where available to check that a required thread is still active.
* Prohibit explicit or implicit calls to program-terminating functions whenever active threads are executing.
* Check and obey the result of stop\_token.stop\_requested() periodically (see 6.60 Concurrency - Directed termination [CGT])
* Return some indication of conditions resulting in premature termination
* Ensure that the main program joins all created threads that need to complete normally, such as by using the std::jthread type.

## 6.64 Uncontrolled Format String [SHL]

### 6.64.1 Applicability to language

The vulnerability as described in ISO/IEC 24772-1 is applicable to C++.

C++ inherits the C libraries which provide a large family of input and output functions that use a control string to interpret the data read or format the output. These strings include all the feature described in ISO/IEC TR 24772-1:2019 clause 6.64.1.

C++ provides type-safe alternatives for input/output, which do not use format strings and which should be used in preference, such as

int aNumber{};  
while(std::cin){ // is input still available  
 std::cout << "Enter a whole number, please:";  
 if (std::cin >> aNumber) { // no format string needed  
 std::cout >> "Thank you, the number can be represented as "  
 std::cout << std::format("0b{0:b} {0:d} 0{0:o} 0x{0:x}", aNumber);  
 } else { // input failed  
 std::cin.clear(); // re-enable input  
 std::string line;  
 getline(std::cin,line); // skip to eol  
 }  
}

In addition, operator overloading of output operators allows to extend formatting abilities to user-defined types.

### 6.64.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects, C++ software developers can:

* Forbid the use of the C library facilities provided by **<cstdio>**.
* Use the avoidance mechanisms provided by the standard library as described in ISO/IEC 14882 20.20 Formatting [format] or similar libraries, if *printf-style* formatting of output or strings is desired, or when the formatting facilities of std::ostream are inadequate.
* Read untrusted input as plain characters and then check for validity, before any conversion to numbers or program entities occurs, using facilities that do not require format strings and that check and report error conditions.

# 6.65 Modifying constants [UJO]

### 6.65.1 Applicability to language

The vulnerability as documented in ISO/IEC TR 24772-1:2019 clause 6.65 exists in C++.

An object can be declared as const, denoting that its value will not change in its lifetime without invoking mechanisms which have undefined behaviour [[EWF]](#EWF). For example, an access path to an object can be declared as const, denoting that the value of the object will not change via this access path without invoking mechanisms which have undefined behaviour, e.g.,

int const i = 0; // the simplest access path  
int& j = const\_cast<int&>(i); // undefined behaviour  
void foo(int\* p) { \*p += 43; }  
//...  
foo(const\_cast<int\*>(&i)); // undefined behaviour  
foo(&i); // ill-formed, compiler error

It is an illegal program or undefined behaviour to attempt to change a const object, such as i, above.

A object that is not const-qualified can be accessed through a path that is const-qualified.

The checking for the correctness of const is enforced based on the access-path and not the type of the target object. While it is possible to remove the const-qualification for an access path, attempting to modify a const object this way is undefined-behavior(see Undefined Behavior [EWF]) : const\_cast<int&> (i) = 0; // undefined behavior

A constant can also be legitimately modified via a secondary access path. For example:

#include <cassert>  
int k = 0;  
void break\_it()   
{  
 k = 42; // legal  
}  
  
void test(int const volatile& j) // Volatile used only to guarantee observability  
{  
 assert(j == 0); // will pass since k == 0  
 break\_it();  
 assert(j == 0); // will fail since k != 0  
}  
  
 test(k);

When using pointers, confusion can occur between qualifications on the pointer’s type (pointer type) and qualifications on the type being referenced (pointee type).

A common misconception is that a member function qualified with const cannot modify any of its members. The following badly defined class introduces a non-const access path to a potentially const object:

struct A  
 {  
 A \* pA;   
 int i{0};   
  
 A () : pA{this}{} // pA provides non-const access path  
  
 void f () const  
 {  
 // pA = nullptr; // ill-formed  
 // i = 0; // ill-formed  
  
 pA->i = 42; // compiles, but undefined behavior  
 // if executed on a const object  
 }  
 };  
   
int main(){  
 A a; // mutable  
 A const b;   
 a.f(); // OK  
 b.f(); // undefined behaviour  
}

However, for C++ classes with members of a pointer-like type, programmers can establish the transitivity of const with const member functions that do not change the referred-to objects.

Within a const member function a mutable data member can still be modified, even if the containing object is const.  
The use of mutable on a data member not contributing to the observable state of the object is preferable to removing the constness of the containing object (see Conversion Errors [FLC]).  
Historically, classes with expensive computations in frequently-called const-member-functions, used mutable data members to cache results of those computations. In concurrent code this practice can lead to data races, unless access to those mutable data members is synchronised.

A safe use of mutable data members is for synchronisation primitives, allowing synchronisation in const member functions.

The following is a common example where a mutex member is declared mutable to allow locking in a const member function:

#include <mutex>  
  
class MyQueue {  
 mutable std::mutex m\_mutex;  
 int \* m\_head { nullptr };  
  
public:  
  
 bool empty () const {  
 std::lock\_guard sg {m\_mutex}; // lock the mutex, which requires m\_mutex to be writable  
 return m\_head != nullptr;  
 }  
  
 // ...  
  
};

### 6.65.2 Avoidance mechanisms for language users

To avoid the vulnerability or mitigate its ill effects, C++ software developers can:

* Use the avoidance mechanisms of ISO/IEC 24772-1 clause 6.65.5.
* Forbid casting away const by const\_cast or by any other means.
* Be aware that, while const means “read only”, C++ permits values accessed by a const access path to be modified via a different access path.
* Use const on pointer-like parameters as a promise to callers and an obligation on the implementation to not modify the referred object.
* Within const member functions, avoid indirectly modifying the object’s state.
* Apply mutable only to member variables that do not influence the observable state of an object.
* Ensure in concurrent code, that mutable data members are synchronisation primitives or are explicitly synchronized.
* When designing C++ classes, provide const member functions to support const objects.
* Ensure that C++ classes modelling pointer-like types provide transitivity of const within const member functions.

# 7. Language specific vulnerabilities for C++

TBD

## 7.1 General

## 7.2 Finalization Issues

### 7.2.1 Description of Vulnerability

Reference 24772-1 clause 7.31 Fault tolerance and failure strategies [REU]

In C++, the finalization of objects, and behaviours such as saving results to external storage before program termination or the notification of other entities external to the program, can be impacted by the way the program terminates.

### 7.2.2 Cross Reference

(MISRA, etc references?)

### 7.2.3 Mechanism of Failure

If std::exit is called during the destruction of an object with static durtion or with thread storage duration, the program has undefined behaviour [[EWF]](#EWF).

If during stack unwinding, the destruction of an object terminates by throwing an exception, the function std::terminate is called, and any subsequent cleanup performed by the program will not occur. Similarly, if the destruction of an object with static duration or with thread storage duration exits via an exception, the function std::terminate is called, and any subsequent cleanup performed by the program will not occur.

Calling std::abort or std::\_Exit terminates the program without executing any destructors and without calling the functions passed to std::atexit or std::at\_quick\_exit. Therefore, any cleanup performed by those functions will not occur. In addition, std::quick\_exit calls std::\_Exit after calling functions passed to std::at\_quick\_exit.

Exceptions thrown in destructors of objects with thread storage duration or in constructors of objects associated with non-block variables with thread storage duration are not caught by a function-try-block on the initial function of the thread or on the main function. This results in a call to std::terminate and any subsequent cleanup performed by the program will not occur.

### 7.2.4 Avoiding the vulnerability or mitigating its effects

* Follow the avoidance mechanisms of [[OYB]][#OYB].
* Avoid calling std::exit from a destructor.
* Avoid throwing an exception from a destructor.
* Avoid calling std::\_Exit or std::abort.
* Avoid calling std::quick\_exit.
* Prefer local objects with static storage duration or with thread storage duration to global objects to ensure that exceptions thrown from their constructors can be caught.

## 7.3 Copy/move semantics from Classes. (Peter Sommerlad’s paper at

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1412r0.pdf

## 7.4 Placement new issues

### 7.4.1 Description of Vulnerability

The C++ object lifetime model allows the creation of an object in existing memory using the (non-allocating) placement new expression. When using this construct the programmer is responsible for ensuring:

* the space requirement of memory for the object;
* that any object previously occupying the space referenced in the placement new expression is properly destroyed before the call;
* the destruction of the new object after use and the return of any space allocated after use;
* that no object is accessed outside of its lifetime; and
* that the storage used by the placement new expression is appropriately destroyed after use.

In addition, it is possible to use a placement new expression to effect type-breaking reinterpretation of data (See [6.11] Pointer type conversion [[HFC]](#HFC)).

### 7.4.2 Cross Reference

### 7.4.3 Mechanism of Failure

If storage is insufficient for the size or alignment of the object being created, then undefined behaviour such as buffer overflow [[HCB]](#HCB) can result.

The lifetime of an object whose storage is reused in a placement new expression is ended, and attempts to access to that object can result in *undefined behaviour*, even though there are a subset of cases where such behaviour is well-defined.

The lifetime of the object whose storage is being replaced is ended, however its destructor is not called implicitly. This can result in *undefined behaviour*, if for example mutexes remain locked or files are left open. Similarly, the lifetime of the newly-created object created by a placement new expression generally requires that its lifetime be managed manually.

A *delete-expression* ends the lifetime of an object by calling the destructor(s), and deallocates the storage. Since a placement new expression calls a non-allocating form of operator new, any attempt to delete the object created by the placement new expression by a delete expression generally results in undefined behaviour.

If the use of a placement new expression results in an overlap with other objects’ storage, then the same issues apply.

A placement new expression without a new-initializer for types with trivial constructors, can result in parts of the object having indeterminate values (likely the values related to the underlying storage).  
This can lead to pointer type reinterpretation issues (see [6.11] Pointer type conversion [[HFC]](#HFC)).

The standard library provides std::construct\_at as an alternative for a placement new expression. The vulnerabilities described above for placement new expressions also apply to std::construct\_at.

### 7.4.4 Avoiding the vulnerability or mitigating its effects

* Apply extreme scrutiny to code that uses placement new expressions or std::construct\_at.

## 7.5 Coroutines

### 7.5.1 Description of Vulnerability

In C++, coroutines are “stackless” suspendable functions which are also implemented in the language in a way that is highly customizable which, when writing code to implement coroutines is suspectible to a number of vulnerabilities including:

* using an object after its ownership has been transferred or its lifetime has ended
* assuming the resumption of execution is in the same context as prior executions or other code
* assuming there will be no use of allocating memory from the heap dynamically

While C++ coroutines can easily be used, they are not as easily defined in a library avoiding vulnerabilities. C++ coroutines involve the following:

* Internal to a coroutine function, all coroutines must use one or more of co\_return, co\_yield, or co\_await, and, the function signature’s types must have a coroutine promise type assocatied with it. Be aware that a coroutine promise type is not the same as or related to std::promise, rather, a coroutine promise type is a type designed to hold coroutine state while the coroutine executes. Coroutines are concurrent yet sequential and do not require threads.
* While the default for C++ coroutines is for the coroutine’s return type to define a suitable coroutine promise type as a member, this can be overridden by defining a std::coroutine\_traits partial specialization. Such specializations must define a promise\_type for a given coroutine function signature. (This implies that the coroutine promise type might not be defined as a member of the coroutine return type.) This allows one to define coroutines that work with others’ types and those other types need not be modified in any way in order for the coroutine to be defined using those types.
* In addition to the coroutine promise type, C++ coroutines are often associated with “awaiter” interfaces (typically implemented as distinct types). Such awaiter interfaces are used with coroutine operations, i.e., co\_await and indirectly with co\_yield and co\_return or operator co\_await.

Authors of the coroutine promise type and awaiter interfaces must write such code with care. Failure to do so can easily result in vulnerabilities.

C++ coroutines are “stackless” which implies:

* Each coroutine call involves only its own stack frame and not the stack frame of all of its enclosing callers. (“Stackful” coroutines in other languages capture the entire stack frame of all enclosing callers. This is not the case in C++: C++ coroutines only have the stack frame of the coroutine itself.)
* Every time a coroutine is resumed, the coroutine stack frame is placed on the executing call stack for that thread before the coroutine is invoked. When a coroutine is suspended the stack frame for the coroutine is then saved (and appropriately removed from the call stack).

Additionally, C++ coroutines need not be resumed on the same thread-of-execution. One major implication of this involves the values in any accessed thread\_local variables which are likely to be different if the thread-of-execution is different. This implies that programmers writing code should not assume values in any thread\_local variable are the same across coroutine suspensions.

Since C++ coroutines can be suspended even before they are initially called the first time, one should not assume a coroutine will be executed since it might never be executed.

Since C++ coroutines can be destroyed or suspended after they have completely finished executing, coroutine library implementers must take care to not access coroutine promise state once a coroutine promise has been destroyed.

C++ coroutine awaiter await\_suspend() code must be written carefully. When await\_suspend() is called the coroutine is already in a suspended state: this function does not suspend the coroutine. Instead await\_suspend() is called on a suspended coroutine and the responsibility of this function is to “schedule” (i.e., to say which coroutine is resumed or destroyed). If a coroutine will be resumed on a different thread-of-execution, changes related to thread execution will occur due to code executed in this function. Since the coroutine is completely suspended while this function executes, there is no race condition concerning whether or not the involved coroutine is executing.

await\_suspend() code should treat \*this as destroyed by not accessing it after the coroutine handle has had its ownership transferred to another object or thread-of-execution. Accessing \*this after such ownership transfers can result in undefined behaviour.

In multi-threaded code sharing state across threads without locks will require suspension code to use (at least) release semantics and resumption code to use (at least) acquire semantics.

In multi-threaded code using locks or acquired resources, any coroutine code that holds a lock or resource when it is suspended, will continue to hold such until it is resumed. Since such can result in issues such as deadlock, it is best to release held locks or resources before suspending.

Generally speaking, objects that manage (one or more) coroutines need to outlive their coroutines’ lifetimes in order to manage them to avoid issues related to their respective lifetimes. Such managing objects will need to wait for the completion of execution of their coroutines so the coroutines do not outlive the objects that manage them. Should a managing object not outlive a coroutine it is managing, those coroutine (promise) states will likely result in memory leaks since they will never be destroyed.

If it is important to control how a coroutine promise is dynamically allocated, then override operator new and operator delete for the promise type. If using a coroutine should never dynamically allocate memory, then define function prototypes for operator new and operator delete and no definition –this will result in the code not being able to be linked if dynamic memory allocation is used.

When await\_resume() is called, coroutine execution does not resume until the call to this function been completed.

### 7.5.2 Cross Reference

### 7.5.3 Mechanism of Failure

In a function that invokes co\_await, co\_yield, and/or co\_return be aware the expression value that is passed to co\_await, co\_yield, and/or co\_return is passed to functions inside that scope. This implies that using these keywords, e.g., co\_yield, their expression values are generated within the scope of the coroutine and are then passed to the appropriate functions inside that scope. For example, consider this non-coroutine function:

std::mutex m;  
int i = 42;  
// ...  
int foo()  
{  
 std::lock\_guard lg{m};  
 return i;  
}

In foo() the mutex m is acquired before accessing the value of i and returning a copy of it. Making that copy occurs while the lock on m is held. After such the function is terminated and all locally-scoped variables are destroyed –which will also release the lock. Contrast such with this:

std::mutex m;  
int i = 42;  
// ...  
std::generator<int> bar()  
{  
 std::lock\_guard lg{m};  
 co\_yield i;  
}

While conceptually one might argue that this code is “equivalent”, in practice, it is different especially in these respects:

* co\_yield does not terminate the coroutine bar() sincebar() is implicitly terminated after it is resumed after the co\_yield suspension has been resumed. (There is an implicit co\_return at the end of this function.)
* If the coroutine promise doesn’t first copy the value of i and uses it directly, e.g., as a reference, and something else concurrently uses that reference but first tries to acquire a lock on m, then that lock acquistion will block until the mutex m is released. The mutex m will not be released until after the coroutine has been resumed but it might never be resumed. The latter would mean that it is possible for the lock on m to result in deadlock.

The above bar() example is therefore better written like this:

std::mutex m;  
int i = 42;  
// ...  
std::generator<int> bar()  
{  
 std::unique\_lock lk{m};  
 auto const tmp{i}; // make a copy  
 lk.unlock(); // release the lock  
 co\_yield tmp; // yield the copy with no locks held  
}

since:

* using i directly requires holding a lock so store i in a local copy which allows releasing the lock
* co\_yield the local copy after releasing the lock so when the coroutine is suspended no lock is held which permits other code needing to acquire that lock to do so

With non-coroutine functions, one typically doesn’t think about this since the non-coroutine function’s execution also ends upon an exception and/or the use of return. The “same” is not true with coroutines:

* a coroutine cannot use return: co\_return must be used
* upon an exception, the coroutine ends so that is effectively the “same” as a non-coroutine function throwing an exception
* upon co\_await or co\_yield the coroutine is typically suspended and not terminated so any locks/resources held by the coroutine at the time of suspension will remain held until the coroutine is resumed which could result in deadlock if the coroutine is never resumed
* upon co\_return, although the coroutine will end soon, any value passed to co\_return occurs while the coroutine is alive and all locks/resources held will continue to be held while such is processed, i.e., by return\_value() in the coroutine promise object (which may or may not be an issue in the coroutine promise object definition)

Additionally, co\_await or co\_yield can also result in values accessible when resumed, e.g.,

// ...  
mycoro misc(int n)  
{  
 for (int i=0; i != n; ++i)  
 i += co\_yield 2\*i;  
}

In this example, i is an int so copies are cheap, small, and easy. If such involves objects that can only be moved, however, if the co\_yielded object is moved to another object (in the coroutine promise type), a copy needs to be made or such cannot again also be moved to the resumed object since the object was already moved away. If the yielding was done with an object that could be copied or moved, then coroutine promise type’s design may need to determine when the object is moved (if ever) or copied since it can only be moved once.

If a coroutine is using any concurrently shared-with-other code locks or resources, then the coroutine will typically need to release such before a suspension to avoid issues such as deadlock. Doing this might require making an additional copy of objects being yielded or returned.

When using coroutines, objects external to the coroutine need to have lifetimes greater than the coroutine’s lifetime regardless of how many times or how long suspensions of the coroutine take. For this reason, it is best to pass and return values and not references to a coroutine, e.g.,

mycoro func1(int i, std::vector<double> v) // OK  
{  
// ...  
}  
  
mycoro func2(int i, std::vector<double>& v) // probably not OK  
{  
// ...  
}  
  
mycoro func3(int i, std::vector<double> const& v) // probably not OK  
{  
// ...  
}

If a coroutine promise’s final\_suspend() function does not return an awaiter that suspends the coroutine, the coroutine state will be immediately destroyed when the coroutine ends and therefore any code that uses coroutine promise state after such will have undefined behaviour. (Typically this will be seen by code exhibiting segmentation faults / access violations trying to access such state.) Should code need to access coroutine promise state after the coroutine has finished executing, ensure that the promise type’s final\_suspend() returns an awaiter that suspends the coroutine, e.g., std::suspend\_always, and anything that manages those coroutine promise instances also destroys those instances when they are no longer needed. Failing to do the latter will result in memory “leaks” of those instances’ promise states (which will never be destroyed).

### 7.5.4 Avoiding the vulnerability or mitigating its effects

* Ensure all objects that manage/own coroutines, outlive the coroutines they manage/own.
* Unless one is absolutely certain the coroutine promise state is no longer required after the coroutine finishes executing, ensure the coroutine promise final\_suspend() returns std::suspend\_always or an awaiter that causes that coroutine to become suspended, and, there is(/are) object(s) that will manage and ultimately destroy those coroutines.
* In await\_suspend() code never access \*this after ownership/control of the coroutine handle has been transferred to other code / threads-of-execution. Instead do what is needed with \*this prior to the ownership/control transfer.

# 8. Implications for standardization

Future standardization efforts should consider:

* Moving in the direction over time to being a more strongly typed
* language. Much of the use of weak typing is simply convenience to the developer in not having to fully consider the types and uses of variables. Stronger typing forces good programming discipline and clarity about variables while at the same time removing many unexpected run time errors due to implicit conversions. This is not to say that C should be strictly a strongly typed language – some advantages of C are due to the flexibility that weaker typing provides. It is suggested that when enforcement of strong typing does not detract from the good flexibility that C offers (for example, adding an integer to a character to step through a sequence of characters) and is only a convenience for programmers (for example, adding an integer to a floating-point number), then the standard should specify the stronger typed solution.
* A common warning in Annex I should be added for floating-point
* expressions being used in a Boolean test for equality.
* Modifying or deprecating many of the C standard library functions
* that make assumptions about the occurrence of a string termination character.
* Define a string construct that does not rely on the null termination
* character.
* Defining an array type that does automatic bounds checking.
* Deprecating less safe functions such as strcpy() and strcat() where
* a more secure alternative is available.
* Defining safer and more secure replacement functions such as
* memncpy() and memncmp() to complement the memcpy() and memcmp() functions (see *6.11.6 Implications for standardization*)
* Defining an array type that does automatic bounds checking.
* Defining functions that contain an extra parameter in memcpy() and
* memmove() for the maximum number of bytes to copy. In the past, some have suggested that the size of the destination buffer be used as an additional parameter. Some critics state that this solution is easy to circumvent by simply repeating the parameter that was used for the number of bytes to copy as the parameter for the size of the destination buffer. This analysis and criticism is correct. What is needed is a failsafe check as to the maximum number of bytes to copy. There are several reasons for creating new functions with an additional parameter. This would make it easier for static analysis to eliminate those cases where the memory copy could not be a problem (such as when the maximum number of bytes is demonstrably less than the capacity of the receiving buffer). Manual analysis or more involved static analysis could then be used for the remaining situations where the size of the destination buffer may not be sufficient for the maximum number of bytes to copy. This extra parameter may also help in determining which copies could take place among objects that overlap. Such copying is undefined according to the C standard. It is suggested that safer versions of functions that include a restriction max\_n on the number of bytes n to copy (for example, void \*memncpy(void \* restrict s1,const void \* restrict s2,size\_t n), const size\_t max\_n) be added to the standard in addition to retaining the current corresponding functions (for example, memcpy(void \* restrict s1,const void \* restrict s2,size\_t n))). The additional parameter would be consistent with the copying function pairs that have already been created such as strcpy()/strncpy() and strcat()/strncat(). This would allow a safer version of memory copying functions for those applications that want to use them in to facilitate both safer and more secure code and more efficient and accurate static code reviews[[1]](#footnote-257).
* Restrictions on pointer arithmetic that could eliminate common
* pitfalls. Pointer arithmetic is error-prone and the flexibility that it offers is useful, but some of the flexibility is simply a shortcut that if restricted could lessen the chance of a pointer arithmetic based error.
* Defining a standard way of declaring an attribute to indicate that a
* variable is intentionally unused.
* A common warning in Annex I should be added for variables with the
* same name in nested scopes.
* Creating a few standardized precedence orders. Standardizing on a
* few precedence orders will help to eliminate the confusing intricacies that exist between languages. This would not affect current languages as altering precedence orders in existing languages is too onerous. However, this would set a basis for the future as new languages are created and adopted. Stating that a language uses “ISO precedence order A” would be useful rather than having to spell out the entire precedence order that differs in a conceptually minor way from some other languages, but in a major way when programmers attempt to switch between languages.
* Deprecating the goto statement. The use of the goto construct is
* often spotlighted as the antithesis of good structured programming. Though its deprecation will not instantly make all C code structured, deprecating the goto and leaving in place the restricted goto variations (for example, break and continue) and possibly adding other restricted goto’s could assist in encouraging safer and more secure C programming in general.
* Defining a "fallthru" construct that will explicitly bind multiple
* switch cases together and eliminate the need for the break statement. The default would be for a case to break instead of falling through to the next case. Granted this is a major shift in concept, but if it could be accomplished, less unintentional errors would occur.
* Defining an identifier type for loop control that cannot be modified
* by anything other than the loop control construct would be a relatively minor addition to C that could make C code safer and encourage better structured programming.
* Defining a standardized interface package for interfacing C with
* many of the top programming languages and a reciprocal package should be developed of the other top languages to interface with C.
* Joining with other languages in developing a standardized set of
* mechanisms for detecting and treating error conditions so that all languages to the extent possible could use them. Note that this does not mean that all languages should use the same mechanisms as there should be a variety ( label parameters, auxiliary status variables), but each of the mechanisms should be standardized.
* Since fault handling and exiting of a program is common to all
* languages, it is suggested that common terminology such as the meaning of fail safe, fail hard, fail soft, and so on along with a core API set such as exit, abort, and so on be standardized and coordinated with other languages.
* Deprecating unions. The primary reason for the use of unions to save
* memory has been diminished considerably as memory has become cheaper and more available. Unions are not statically type safe and are historically known to be a common source of errors, leading to many C programming guidelines specifically prohibiting the use of unions.
* Creating a recognizable naming standard for routines such that one
* version of a library does parameter checking to the extent possible and another version does no parameter checking. The first version would be considered safer and more secure and the second could be used in certain situations where performance is critical and the checking is assumed to be done in the calling routine. A naming standard could be made such that the library that does parameter checking could be named as usual, say “library\_xyz” and an equivalent version that does not do checking could have a “\_p” appended, such as “library\_xyz\_p”. Without a naming standard such as this, a considerable number of wasted cycles will be conducted doing a double check of parameters or even worse, no checking will be done in both the calling and receiving routines as each is assuming the other is doing the checking.
* Creating an Annex that lists deprecated features.

# Bibliography

[1] ISO/IEC Directives, Part 2, *Rules for the structure and drafting of International Standards*, 2004

[2] ISO/IEC TR 10000‑1, *Information technology — Framework and taxonomy of International Standardized Profiles — Part 1: General principles and documentation framework*

[3] ISO 10241 (all parts), *International terminology standards*

[4] ISO/IEC 9899:2011, *Information technology* — *Programming languages —* C

[5] ISO/IEC 9899:2011/Cor.1:2012, *Technical Corrigendum 1*

[6] ISO/IEC 30170:2012, *Information technology* — *Programming languages —* Ruby

[7] ISO/IEC/IEEE 60559:2011, *Information technology – Microprocessor Systems – Floating-Point arithmetic*

[8] ISO/IEC 1539-1:2010, *Information technology — Programming languages —* Fortran — Part 1: Base language

[9] ISO/IEC 8652:1995, *Information technology — Programming languages —* Ada

[10] ISO/IEC 14882:2011, *Information technology* — *Programming languages —* C++

[11] R. Seacord, *The CERT C Secure Coding Standard*. Boston,MA: Addison-Westley, 2008.

[12] Motor Industry Software Reliability Association. *Guidelines for the Use of the C Language in Vehicle Based Software*, 2012 (third edition)16F[[2]](#footnote-259).

[13] ISO/IEC TR24731–1, *Information technology — Programming languages, their environments and system software interfaces — Extensions to the C library — Part 1: Bounds-checking interfaces*

[14] ISO/IEC TR 15942:2000, *Information technology — Programming languages — Guide for the use of the Ada programming language in high integrity systems*

[15] Joint Strike Fighter Air Vehicle: C++ Coding Standards for the System Development and Demonstration Program. Lockheed Martin Corporation. December 2005.

[16] Motor Industry Software Reliability Association. *Guidelines for the Use of the C++ Language in critical systems*, June 2008

[17] ISO/IEC TR 24718: 2005, *Information technology — Programming languages — Guide for the use of the Ada Ravenscar Profile in high integrity systems*

[18] L. Hatton, Safer C: developing software for high-integrity and safety-critical systems. McGraw-Hill 1995

[19] ISO/IEC 15291:1999, *Information technology — Programming languages — Ada Semantic Interface Specification (ASIS)*

[20] Software Considerations in Airborne Systems and Equipment Certification. Issued in the USA by the Requirements and Technical Concepts for Aviation (document RTCA SC167/DO-178B) and in Europe by the European Organization for Civil Aviation Electronics (EUROCAE document ED-12B).December 1992.

[21] IEC 61508: Parts 1-7, Functional safety: safety-related systems. 1998. (Part 3 is concerned with software).

[22] ISO/IEC 15408: 1999 Information technology. Security techniques. Evaluation criteria for IT security.

[23] J Barnes, High Integrity Software - the SPARK Approach to Safety and Security. Addison-Wesley. 2002.

[25] Steve Christy, *Vulnerability Type Distributions in CVE*, V1.0, 2006/10/04

[26] *ARIANE 5: Flight 501 Failure*, Report by the Inquiry Board, July 19, 1996 <http://esamultimedia.esa.int/docs/esa-x-1819eng.pdf>

[27] Hogaboom, Richard, *A Generic API Bit Manipulation in C*, Embedded Systems Programming, Vol 12, No 7, July 1999 <http://www.embedded.com/1999/9907/9907feat2.htm>

[28] Carlo Ghezzi and Mehdi Jazayeri, Programming Language Concepts, 3rd edition, ISBN-0-471-10426-4, John Wiley & Sons, 1998

[29] Lions, J. L. [ARIANE 5 Flight 501 Failure Report](http://en.wikisource.org/wiki/Ariane_501_Inquiry_Board_report). Paris, France: European Space Agency (ESA) & National Center for Space Study (CNES) Inquiry Board, July 1996.

[30] Seacord, R. *Secure Coding in C and C++*. Boston, MA: Addison-Wesley, 2005. See <http://www.cert.org/books/secure-coding> for news and errata.

[31] John David N. Dionisio. Type Checking. <http://myweb.lmu.edu/dondi/share/pl/type-checking-v02.pdf>

[32] MISRA Limited. "[MISRA C](http://www.misra.org.uk/): 2012 Guidelines for the Use of the C Language in Critical Systems." Warwickshire, UK: MIRA Limited, March 2013 (ISBN 978-1-906400-10-1 and 978-1-906400-11-8).

[33] The Common Weakness Enumeration (CWE) Initiative, MITRE Corporation, (<http://cwe.mitre.org/>)

[34] Goldberg, David, *What Every Computer Scientist Should Know About Floating-Point Arithmetic*, ACM Computing Surveys, vol 23, issue 1 (March 1991), ISSN 0360-0300, pp 5-48.

[35] IEEE Standards Committee 754. IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-2008. Institute of Electrical and Electronics Engineers, New York, 2008.

[36] Robert W. Sebesta, Concepts of Programming Languages, 8th edition, ISBN-13: 978-0-321-49362-0, ISBN-10: 0-321-49362-1, Pearson Education, Boston, MA, 2008

[37] Bo Einarsson, ed. Accuracy and Reliability in Scientific Computing, SIAM, July 2005 <http://www.nsc.liu.se/wg25/book>

[38] GAO Report, Patriot *Missile Defense: Software Problem Led to System Failure at Dhahran, Saudi Arabia*, B-247094, Feb. 4, 1992, <http://archive.gao.gov/t2pbat6/145960.pdf>

[39] Robert Skeel, *Roundoff Error Cripples Patriot Missile*, SIAM News, Volume 25, Number 4, July 1992, page 11, <http://www.siam.org/siamnews/general/patriot.htm>

[40] CERT. *CERT C++ Secure Coding Standard*.  [https://www.securecoding.cert.org/confluence/pages/viewpage.action?pageId=637](https://www.securecoding.cert.org/confluence/pages/viewpage.action?pageId=637%20) (2009).

[41] Holzmann, Garard J., Computer, vol. 39, no. 6, pp 95-97, Jun., 2006, *The Power of 10: Rules for Developing Safety-Critical Code*

[42] P. V. Bhansali, A systematic approach to identifying a safe subset for safety-critical software, ACM SIGSOFT Software Engineering Notes, v.28 n.4, July 2003

[43] Ada 95 Quality and Style Guide, SPC-91061-CMC, version 02.01.01. Herndon, Virginia: Software Productivity Consortium, 1992. Available from: <http://www.adaic.org/docs/95style/95style.pdf>

[44] Ghassan, A., & Alkadi, I. (2003). Application of a Revised DIT Metric to Redesign an OO Design. *Journal of Object Technology* , 127-134.

[45] Subramanian, S., Tsai, W.-T., & Rayadurgam, S. (1998). Design Constraint Violation Detection in Safety-Critical Systems. The 3rd IEEE International Symposium on High-Assurance Systems Engineering , 109 - 116.

[46] Lundqvist, K and Asplund, L., “*A Formal Model of a Run-Time Kernel for Ravenscar*”, The 6th International Conference on Real-Time Computing Systems and Applications – RTCSA 1999

[47] ISO/IEC TS 17961, *Information technology – Programming languages, their environments and system software interfaces – C secure coding rules*

[48] GNU Project. GCC Bugs “Non-bugs” http://gcc.gnu.org/bugs.html#nonbugs\_c (2009).

# Index

LHS (left-hand side), 22

1. ::: {custom-style=“footnote text”} This has been addressed by WG 14 in an optionally normative annex in the current working paper [↑](#footnote-ref-257)
2. ::: {custom-style=“footnote text”} The first edition should not be used or quoted in this work. [↑](#footnote-ref-259)