ISO/IEC JTC 1/SC 22/WG23 N0838

Date: 2018-11-07

ISO/IEC TR 24772–9

Notes on this document

This document is an early draft of a Guidance to avoiding programming language vulnerabilities in C++. It started its existence as a direct copy from the equivalent C language document, with the intention to replace the C subclauses with ones that are relevant to C++.

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

* 6.3 Bit representation
* 6.5 Enumerator issues [CCB],
* 6.6 Conversion errors
* 6.7 String termination
* 6.8 Buffer boundary violation
* 6.9 Unchecked array indexing
* 6.11 Pointer type conversions
* 6.10 Unchecked array copying (needs to be revisited)
* 6.13 Null pointer dereference [XYH],
* 6.19 Unused variables
* 6.22 Initialization of variables [LAV],
* 6.26 Dead store,
* 27 Switch statements and static analysis
* 28 Demarcation of control flow
* 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 Ingored 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
* 6.40 Templates and generics (1/2 done)
* 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

Edition 1

ISO/IEC JTC 1/SC 22/WG 23

Secretariat: ANSI

Information Technology — Programming languages — Guidance to avoiding vulnerabilities in programming languages – Part 9 – 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](#_Toc1165219)

[Introduction viii](#_Toc1165220)

[1. Scope 1](#_Toc1165221)

[2. Normative references 1](#_Toc1165222)

[3. Terms and definitions, symbols and conventions 1](#_Toc1165223)

[3.1 Terms and definitions 1](#_Toc1165224)

[4. Language concepts 4](#_Toc1165225)

[5. Avoiding programming language vulnerabilities in C++ 4](#_Toc1165226)

[6. Specific Guidance for C++ Vulnerabilities 6](#_Toc1165227)

[6.1 General 6](#_Toc1165228)

[6.2 Type System [IHN] 6](#_Toc1165229)

[6.3 Bit Representations [STR] 7](#_Toc1165230)

[6.4 Floating-point Arithmetic [PLF] 8](#_Toc1165231)

[6.5 Enumerator Issues [CCB] 8](#_Toc1165232)

[6.6 Conversion Errors [FLC] 9](#_Toc1165233)

[6.7 String Termination [CJM] 10](#_Toc1165234)

[ Use std::string or similar, in preference to C-style arrays of chars 11](#_Toc1165235)

[6.8 Buffer Boundary Violation [HCB] 11](#_Toc1165236)

[6.9 Unchecked Array Indexing [XYZ] 12](#_Toc1165237)

[6.10 Unchecked Array Copying [XYW] 13](#_Toc1165238)

[6.11 Pointer Type Conversions [HFC] 13](#_Toc1165239)

[6.12 Pointer Arithmetic [RVG] 15](#_Toc1165240)

[6.13 NULL Pointer Dereference [XYH] 16](#_Toc1165241)

[6.14 Dangling Reference to Heap [XYK] 16](#_Toc1165242)

[6.15 Arithmetic Wrap-around Error [FIF] 18](#_Toc1165243)

[6.16 Using Shift Operations for Multiplication and Division [PIK] 18](#_Toc1165244)

[6.17 Choice of Clear Names [NAI] 19](#_Toc1165245)

[6.18 Dead Store [WXQ] 19](#_Toc1165246)

[6.19 Unused Variable [YZS] 20](#_Toc1165247)

[6.20 Identifier Name Reuse [YOW] 20](#_Toc1165248)

[6.21 Namespace Issues [BJL] 21](#_Toc1165249)

[6.22 Initialization of Variables [LAV] 21](#_Toc1165250)

[6.23 Operator Precedence and Associativity [JCW] 21](#_Toc1165251)

[6.24 Side-effects and Order of Evaluation of Operands [SAM] 21](#_Toc1165252)

[6.25 Likely Incorrect Expression [KOA] 22](#_Toc1165253)

[6.26 Dead and Deactivated Code [XYQ] 24](#_Toc1165254)

[6.27 Switch Statements and Static Analysis [CLL] 24](#_Toc1165255)

[6.28 Demarcation of Control Flow [EOJ] 25](#_Toc1165256)

[6.29 Loop Control Variables [TEX] 26](#_Toc1165257)

[6.30 Off-by-one Error [XZH] 27](#_Toc1165258)

[6.31 Structured Programming [EWD] 28](#_Toc1165259)

[6.32 Passing Parameters and Return Values [CSJ] 28](#_Toc1165260)

[6.33 Dangling References to Stack Frames [DCM] 29](#_Toc1165261)

[6.34 Subprogram Signature Mismatch [OTR] 30](#_Toc1165262)

[6.35 Recursion [GDL] 31](#_Toc1165263)

[6.36 Ignored Error Status and Unhandled Exceptions [OYB] 31](#_Toc1165264)

[6.37 Type-breaking Reinterpretation of Data [AMV] 32](#_Toc1165265)

[6.38 Deep vs. Shallow Copying [YAN] 33](#_Toc1165266)

[6.39 Memory Leak and Heap Fragmentation [XYL] 33](#_Toc1165267)

[6.40 Templates and Generics [SYM] 34](#_Toc1165268)

[6.41 Inheritance [RIP] 35](#_Toc1165269)

[6.41.1 Applicability to language 35](#_Toc1165270)

[6.41.2 Guidance to language users 37](#_Toc1165271)

[6.42 Violations of the Liskov Substitution Principle or the Contract Model [BLP] 37](#_Toc1165272)

[6.42.1 Applicability to language 37](#_Toc1165273)

[6.42.2 Guidance to language users 38](#_Toc1165274)

[6.43 Redispatching [PPH] 38](#_Toc1165275)

[6.43.1 Applicability to language 38](#_Toc1165276)

[6.43.2 Guidance to language users 39](#_Toc1165277)

[6.44 Polymorphic variables [BKK] 39](#_Toc1165278)

[6.44.1 Applicability to language 40](#_Toc1165279)

[6.44.2 Guidance to language users 41](#_Toc1165280)

[6.45 Extra Intrinsics [LRM] 42](#_Toc1165281)

[6.46 Argument Passing to Library Functions [TRJ] 42](#_Toc1165282)

[6.47 Inter-language Calling [DJS] 42](#_Toc1165283)

[6.48 Dynamically-linked Code and Self-modifying Code [NYY] 45](#_Toc1165284)

[6.49 Library Signature [NSQ] 46](#_Toc1165285)

[6.50 Unanticipated Exceptions from Library Routines [HJW] 47](#_Toc1165286)

[6.51 Pre-processor Directives [NMP] 48](#_Toc1165287)

[6.52 Suppression of Language-defined Run-time Checking [MXB] 49](#_Toc1165288)

[6.53 Provision of Inherently Unsafe Operations [SKL] 49](#_Toc1165289)

[6.54 Obscure Language Features [BRS] 49](#_Toc1165290)

[6.55 Unspecified Behaviour [BQF] 50](#_Toc1165291)

[6.56 Undefined Behaviour [EWF] 50](#_Toc1165292)

[6.57 Implementation–defined Behaviour [FAB] 51](#_Toc1165293)

[6.58 Deprecated Language Features [MEM] 52](#_Toc1165294)

[6.59 Concurrency – Activation [CGA] 52](#_Toc1165295)

[6.60 Concurrency – Directed termination [CGT] 52](#_Toc1165296)

[6.60.1 Applicability to language 53](#_Toc1165297)

[6.60.2 Guidance to language users 53](#_Toc1165298)

[6.61 Concurrent Data Access [CGX] 53](#_Toc1165299)

[6.62 Concurrency – Premature Termination [CGS] 53](#_Toc1165300)

[6.63 Protocol Lock Errors [CGM] 53](#_Toc1165301)

[6.64 Uncontrolled Format String [SHL] 54](#_Toc1165302)

[7. Language specific vulnerabilities for C 54](#_Toc1165303)

[8. Implications for standardization 54](#_Toc1165304)

[Bibliography 57](#_Toc1165305)

[Index 60](#_Toc1165306)

# 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-X, 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 considering C++ or using 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.

Abstract

Access protection

Concrete

Class

Dynamic dispatch

Encapsulation

Inheritance

Namespace

Overload

Override

Protected

Private

Public

Pure

Static

STL

Template

Virtual

3.1.1

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

**3.1.2**

**alignment**
The 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.

**3.1.4**

**behaviour**
An external appearance or action.

Note 1: See: implementation-defined behavior, locale-specific behavior, undefined behavior, unspecified behaviour

**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. It need not be possible to express the address of each individual bit of an object.

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

Note 1: 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.

**character**
 An abstract member of a set of elements used for the organization, control, or representation of data.

Note 6: See: single-byte character, multibyte character, wide character

correctly rounded result: The 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.

diagnostic message: The message belonging to an implementation-defined subset of the implementation’s message output. The C Standard requires diagnostic messages for all constraint violations.

formal parameter: The 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.

implementation: A particular set of software, running in a particular translation environment under particular control options, that performs translation of programs for, and supports execution of functions in, a particular execution environment.

implementation-defined behaviour: The unspecified behaviour where each implementation documents how the choice is made. An example of implementation-defined behaviour is the propagation of the high-order bit when a signed integer is shifted right.

implementation-defined value: An unspecified value where each implementation documents how the choice for the value is selected.

implementation limit: The restriction imposed upon programs by the implementation.

indeterminate value: Is either an unspecified value or a trap representation.

Language type: See block-structured language, comb-structured language

locale-specific behaviour: The behaviour that depends on local conventions of nationality, culture, and language that each implementation documents. An example, locale-specific behaviour is whether the islower() function returns true for characters other than the 26 lower case Latin letters.

memory location: Either an object of scalar[[1]](#footnote-1) type, or a maximal sequence of adjacent bit-fields all having nonzero width.

Note 1: A bit-field and an adjacent non-bit-field member are in separate memory locations. The same applies to two bit-fields, if one is declared inside a nested structure declaration and the other is not, or if the two are separated by a zero-length bit-field declaration, or if they are separated by a non-bit-field member declaration. It is not safe to concurrently update two bit-fields in the same structure if all members declared between them are also bit-fields, no matter what the sizes of those intervening bit-fields happen to be. For example a structure declared as

 struct {

 char a;

 int b:5, c:11, :0, d:8;

 struct { int ee:8; } e;

 }

contains four separate memory locations: The member a, and bit-fields d and e.ee are separate memory locations, and can be modified concurrently without interfering with each other. The bit-fields b and c together constitute the fourth memory location. The bit-fields b and c can’t be concurrently modified, but b and a, can be concurrently modified.

multibyte character: The sequence of one or more bytes representing a member of the extended character set of either the source or the execution environment. The extended character set is a superset of the basic character set.

object: The region of data storage in the execution environment, the contents of which can represent values. When referenced, an object may be interpreted as having a particular type.

parameter: See actual argument, argument, formal parameter

recommended practice: A specification that is strongly recommended as being in keeping with the intent of the C Standard, but that may be impractical for some implementations.

runtime-constraint: A requirement on a program when calling a library function.

single-byte character: The bit representation that fits in a byte.

trap representation: An object representation that need not represent a value of the object type.

undefined behaviour: The use of a non-portable or erroneous program construct or of erroneous data, for which the C standard imposes no requirements. 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.

unspecified behaviour: The use of an unspecified value, or other behaviour where the C Standard provides two or more possibilities and imposes no further requirements on which is chosen in any instance. For example, unspecified behaviour is the order in which the arguments to a function are evaluated.

unspecified value: The valid value of the relevant type where the C Standard imposes no requirements on which value is chosen in any instance. An unspecified value cannot be a trap representation.

value: The precise meaning of the contents of an object when interpreted as having a specific type. See implementation-defined value, indeterminate value, unspecified value, trap representation

wide character: A bit representation capable of representing any character in the current locale. The C Standard uses the name wchar\_t for objects of this type.

# 4. Language concepts

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

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*

# 5. Avoiding programming language vulnerabilities in C++

In addition to the generic programming rules from TR 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 |
| 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.

## 6.2 Type System [IHN]

### 6.2.1 Applicability to language

AI –

Ideas (Much of this can go to language concepts)

* C++ is a rich language (rich type system) with many nuances. 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” keyword for constructors and conversion operators
* operator bool() 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).

The primitive numeric types of C++, for historical reasons, allow a variety of implicit conversions, some of which are unsafe. C++ class types, in contrast, have strictly limited implicit operations and conversions, and may practically be used in place of primitive numeric types. 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++ Dynamic cast and the use of it during construction and deconstruction needs further exposition. The this pointer type can have surprising effects.

References

* CERT section OOP (AI – Aaron to provide others), (note that some of these will likely migrate to other vulnerabilities)x
	+ DCL52-CPP. Never qualify a reference type with const or volatile
	(this one is odd because the language makes this an error, but some
	compilers like MSVC only warn on it, but will still translate the
	source somehow)
	+ DCL60-CPP. Obey the one-definition rule
	+ DCL40-C. Do not create incompatible declarations of the same function or object
	+ EXP51-CPP. Do not delete an array through a pointer of the incorrect type
	+ EXP55-CPP. Do not access a cv-qualified object through a cv-unqualified type
	+ EXP56-CPP. Do not call a function with a mismatched language linkage
	+ EXP57-CPP. Do not cast or delete pointers to incomplete classes
	+ EXP60-CPP. Do not pass a nonstandard-layout type object across
	execution boundaries
	+ EXP36-C. Do not cast pointers into more strictly aligned pointer types
	+ EXP47-C. Do not call va\_arg with an argument of the incorrect type
	+ OOP51-CPP. Do not slice derived objects
	+ OOP52-CPP. Do not delete a polymorphic object without a virtual destructor
* AI – Lisa – look at C++ Core Guidelines for “casts”
	+ ES48 avoid casts
	+ ES49 if using a cast, use a named cast
	+ ES50 don’t cast away const
* C++ Core guidelines for conversions
	+ ES23 prefer {}
	+ ES46 Avoid narrowing conversions
	+ ES64 use T{e} notation for construction
	+ ES100 don’t mix signed and unsigned arithmetic
	+ ES103 Don’t overflow
	+ ES104 Don’t underflow (really overflow negatively)
* AUTOSAR (AI Peter to work with AUTOSAR to provide references)

### 6.2.2 Guidance to language users

For specific types discussed in this document, such as floating point types, see the respective clauses.

* Treat every explicit cast as a candidate for refactoring.
* Use C++ casts rather than C-style casts, as they provide more compile-time checking and are more restrictive in what they can change.
* *Class member functions that can be ‘static’ should be ‘static’. Class member functions that cannot be ‘static’, but can be ‘const’ should be ‘const’*
* *The ‘mutable’ keyword for class member variables should be used sparingly*
* Do not use volatile for inter-thread communication or synchronization
	+ See C++ Core guidelines CP.8, CP.200, CP.111,
* Don't mix signed and unsigned types in arithmetic
* Follow the advice provided in TR 24772-3 clause 6.2.2. when using C-style numeric types, and implicit conversions.

## 6.3 Bit Representations [STR]

### 6.3.1 Applicability to language

This vulnerabilities described in TR24772-1 clause 6.3 is applicable to C++.

Document the C++ behaviours- handling bit-fields, - hitting enclosing word, concurrent access, hardware implications,

Able to use non-integer types (such as enumerations) in accessing bit fields.

A C++ memory location is either an object is or a contiguous collection of bit-fields.

C++ bit fields are not separated from adjacent bit-fields for purposes of thread synchronization or volatility. Bit-fields are very difficult to use correctly in these contexts.

**6.3.2 Guidance to language users**

In addition to the advice of TR 24772-3 clause 6.3.2:

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

CERT INT34-C

* Do not use std::vector<bool>
* Use bit-fields with care or avoid them entirely. Instead, use a class type containing one or more unsigned integer data members and member functions appropriate to the particular situation.
* Do not create a bit-field of a signed type and size one.

See AUTOSAR A9-6-1,

Issue was raised about padding bits between object/struct/union members can leak information. Where to put this? Mitigation – use member copy instead of byte-wise copy.

CERT EXP62-CPP

## 6.4 Floating-point Arithmetic [PLF]

### 6.4.1 Applicability to language

C++ uses the floating point mechanisms of C, as documented in TR 24772-3 clause 6.4.1.

AI – steve – speak with Hubert about C++ FP issues and see what needs to be done.

### 6.4.2 Guidance to language users

Follow the general advice of TR 24772-3 clause 6.4.2.

## 6.5 Enumerator Issues [CCB]

6.5.1 Applicability to language

### 6.5.1.1 References

AUTOSAR A7-2-2 Enumeration base type shall be explicitly defined

6.5.1.2 **Applicability**

C++ offers enums for defining distinct types composed of sets of related named constants. The type of each enum is different from all other types. Each enum has an underlying integral type, which the user can specify. Since enums are distinct types, the user can only assign values to an object of enumerated type that are values of that enumerated type. C++ does not support implicit conversion of an int to an enum, therefore preventing A = B + C where A, B and C are variables of the same enum.

C++ enums can be scoped (enum class) or unscoped (enum). C++ supports implicit conversion of an unscoped enum to an integer by integral promotion

enum Color {red, green, blue};

int i = red; // implicit conversion

C++ does not support implicit conversion of a scoped enum to an int. Hence, operations such as ++, +, < and enums used as array indices require explicit definitions.

enum class Color {red, green, blue};

int i = red; // error – no implicit conversion

Where unscoped enums are used as array indexes and have a user-specified mapping to an underlying representation, there will be “holes” as documented in TR24772-1 clause 6.6.

Scoped enum types cannot be used as the index of an array.

In C++ 2017, casting a value to an enumeration type is undefined behavior unless the source value is within the range of values of an enumeration type. See CERT INT50-CPP.

### 6.5.2 Guidance to language users

* Use *scoped enumerations* in preference tothe C-style *unscoped enumerations* for related values.
	+ See CPP Core Guidelines Enum.4 and Enum.6 (titles?)
	+ See AUTOSAR A7-2-3 “Enumerations shall be declared as scoped enum classes”
* Use constexpr to declare a set of unrelated values, such as
constexpr size\_t bufferLen = 128;
constexpr char special\_char = ‘a’;
* If *unscoped enumerations* are used, follow the general advice of TR 24772-3 clause 6.5.2 as well as the following:
* Avoid casting arbitrary integer values to enumeration type. If it is unavoidable, use braced initialization instead of C-style or static casts
 e\_type{7};
	+ See CERT INT50-CPP Do no Cast to an out-of-range-value
* Obtain the underlying enumeration value, by casting the enumeration to its underlying type, e.g.,

enum e\_type{A, B, C};

auto value = static\_cast<std::underlying\_type\_t<e\_type>>(B);

## 6.6 Conversion Errors [FLC]

### 6.6.1 Applicability to language

C++ includes some of the conversion mechanisms of C, as documented in TR 24772-3 clause 6.6.1.

C++ type conversion mechanisms differ from the mechanisms of C, as documented in ISO IEC 14882 Annex C. This subclause highlights those differences where C++ eliminates potential vulnerabilities found in C.

Implicit conversions from void\* to any other object type is invalid.

C++ adds a number of new features relevant to type conversion:

* C-style casts (using the desired type in brackets in front of an expression), whilst still available in C++, are augmented by four C++ specific cast and function style casts. These provide a number of (mostly) compile-time checks, so prevent casting between obviously inappropriate types
* The programmer can add code to the definition of a class to allow values of any other type to be implicitly cast to that class type, or for a class object to be implicitly cast to any other type (including basic numeric types). As implicit conversions can make code maintenance more difficult, in general they should be avoided

Implicit casting to a class type occurs when a class has a constructor that can take a single parameter, as in the following example:

class C

 {public:

 C(int x=10, float y=0){…}

 };

void foo(C param){…}

… foo(21); …

The call to foo requires a parameter of type C, but is provided with an int. However, as C has a constructor that can take an int parameter (the float parameter is ignored because it has a default value), a temporary object of type C is constructed using 21 as the x parameter. This is passed to foo. The temporary object is destroyed when foo returns.

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

 explicit C(int x=10, float y=0){…}

The call foo(21) would now not be legal.

### 6.6.2 Guidance to language users

In addition to the general advice of TR 24772-1 clause 6.6.5:

* Guidance for numeric conversions: Use the brace form of function style casts
* Use C++ casts rather than C-style casts, as they provide more checking
* If a class has a converting constructor and implicit conversions are not required, make that constructor ‘explicit’

## 6.7 String Termination [CJM]

### 6.7.1 Applicability to language

The vulnerability as documented in TR 24772-1 exists in C++ when C-style strings are used. A string in C++ is composed of a contiguous sequence of characters terminated by and including a null character (a byte with all bits set to 0). Therefore strings in C++ cannot contain the null character except as the terminating character. Inserting a null character in a string either through a bug or through malicious action can truncate a string unexpectedly. Alternatively, not putting a null character terminator in a string can cause actions such as string copies to continue well beyond the end of the expected string. Overflowing a string buffer through the intentional lack of a null terminating character can be used to expose information or to execute malicious code.

In C, strings are usually implemented as arrays of chars. Such arrays can be prone to accidental or deliberate overflow, as they are inherently of a fixed size. Hence attempting to copy an string longer than the array, or appending a string where the result will be longer than the array, will lead to corruption of the program state.

C++ provide a string class (in the iostream library), std::string. Internally, the class maintains an array of char on the heap. If an attempt is made to copy or append a string that results in a string larger than the current size of the array, a new larger array is allocated.

UNICODE and multibyte strings??

### 6.7.2 Guidance to language users

## Use std::string or similar, in preference to C-style arrays of chars

## 6.8 Buffer Boundary Violation [HCB]

### 6.8.1 Applicability to language

A buffer boundary violation condition occurs when an array is indexed outside its bounds, or pointer arithmetic results in an access to storage that occurs outside the bounds of the object accessed. This behaviour may occur when copying, initializing, writing or reading.

In C++, the built-in subscript operator [] is defined such that E1[E2] is identical to (\*((E1)+(E2))), so that in either representation, the value in location (E1+E2) is returned. C++ does not perform bounds checking on arrays: arrays may be accessed outside of their bounds which is undefined behaviour. For example, in C++ the following code is syntactically valid, though, if offset has the value 10, the behaviour is undefined:

int foo(const int offset) {

 int t;

 int x[] = {0,0,0,0,0};

 t = x[offset];

 return t;

 }

or, when written using iterators, the same issues can occur

int foo(const int offset) {

  std::array<int, 5> a;

  return \*(a.begin() + offset);

For further explanation and examples, see

[https://wiki.sei.cmu.edu/confluence/display/cplusplus/CTR50-CPP.+Guarantee+that+container+indices+and+iterators+are+within+the+valid+range](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CTR50-CPP.%2BGuarantee%2Bthat%2Bcontainer%2Bindices%2Band%2Biterators%2Bare%2Bwithin%2Bthe%2Bvalid%2Brange)

[https://wiki.sei.cmu.edu/confluence/display/cplusplus/CTR53-CPP.+Use+valid+iterator+ranges](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CTR53-CPP.%2BUse%2Bvalid%2Biterator%2Branges)

[https://wiki.sei.cmu.edu/confluence/display/cplusplus/CTR55-CPP.+Do+not+use+an+additive+operator+on+an+iterator+if+the+result+would+overflow](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CTR55-CPP.%2BDo%2Bnot%2Buse%2Ban%2Badditive%2Boperator%2Bon%2Ban%2Biterator%2Bif%2Bthe%2Bresult%2Bwould%2Boverflow)

Note: Consider C++ Core guidelines if completed.

*As described in 6.7 [CJM], C++ provides library functions, e.g. std::string, that encapsulate strings and prevent boundary violations when accessing arrays of characters. It also provides standard templates that provide similar facilities for any other type, such as std::vector.*

### 6.8.2 Guidance to language users

* Avoid C-style arrays. Guidance for the use of C-style arrays is provided in TR 24772-3 clause 6.8.2.
* Use a library class such as std::array to encapsulate an array, or write a class with similar behavior.
* Use containers of the standard library, such as std::vector or std::deque, to model arrays with dynamically changing size.
* Use iterator-based algorithms, such as those of the standard library.
* Use the range-based for loop construct such as for (auto I: *some container*) to iterate within the defined bounds of the object.
* Use iterators over the range of elements to be accessed instead of using an array and bounds as parameters.
* Perform range checking before indexing into an array. In the interest of speed and efficiency, range checking only needs to be done when it cannot be statically shown that an access outside of the array cannot occur.
* When performing random access by indexing, follow the guidance of clause 6.9.2. When performing other forms of random access, follow the guidance of clause 6.12.2. *(Define random access in clause 3 or 4)*
* Use static analysis tools to detect buffer boundary violations.

## 6.9 Unchecked Array Indexing [XYZ]

### 6.9.1 Applicability to language

Like a C-style array, some STL containers, such as std::vector, can be indexed using [], and as in C such an access is unchecked. However, these containers also provide an access function at() that behaves like [], but performs a check that the access is within the bounds of the container.

Similar issues arise from accessing elements in containers by pointer arithmetic.

The following example compares C and C++ performing equivalent array operations:

|  |  |  |
| --- | --- | --- |
| **C** | **C++** | **Comment** |
|  | #include <array> |  |
| int arr [10]; | std::array<int,10>arr; | Both arrays are of 10 elements |
| arr[10] = 0; | arr[10] = 0; | Both accesses silently violate array’s bounds |
| arr[10] = 0; | arr.at(10) = 0; | The C++ access fails with an error exception |

6.9.2 Guidance to language users

* Follow the guidance from clause 6.8.2.
* Use static analysis or explicit checks to establish that bounds violations do not occur. Otherwise use the at() member function of the standard library containers and handle the bounds violation exceptions. See clause 6.36 Ignored error status and unhandled exceptions.

## 6.10 Unchecked Array Copying [XYW]

### 6.10.1 Applicability to language

This subclause requires a complete rewrite.

A buffer overflow occurs when some number of bytes (or other units of storage) is copied from one buffer to another and the amount being copied is greater than is allocated for the destination buffer. In essence this is a special case of Buffer Boundary Violation [HCB].

As with [HCB], in most cases the vulnerability can be avoided by using library classes, such as std::vector, which provides a copy assignment operator that adjusts the size of the target to fit the object being copied.

If for some reason this is not acceptable, C++ has access to the C library functions memcpy and memmove. Both simply copy memory and no checks are made as to whether the destination area is large enough to accommodate the amount of data being copied. It is assumed that the calling routine or programmer has ensured that adequate space has been provided in the destination. Problems can arise when the destination buffer is too small to receive the amount of data being copied.

### 6.10.2 Guidance to language users

This subclause requires a complete rewrite.

* Use standard library containers, such as std::vector, that provide copying mechanisms that ensure the target array is large enough for the indicated source.
* For copies of fixed-sized arrays, perform range checking to prevent out-of-bounds access on the target and the source arrays. In the interest of speed and efficiency, range checking only needs to be done when it cannot be statically shown that an access outside of the arrays cannot occur.
* Use std::string\_view to represent immutable string literals.
* Use std:string to represent mutable strings.

## 6.11 Pointer Type Conversions [HFC]

### 6.11.1 Applicability to language

In this clause, all C++ references, in addition to pointers. The shared\_ptr casts

The vulnerabilites as described in TR 24772-1 clause 6.11.1 also apply to C++.

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 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.

Reinterpret\_cast has the problem that it simply treats the unmodified pattern of bits in the pointer as being of the target type rather than the original type, but the C++ standard recognizes that the language or compiler may impose constraints or additional data requirements on a pointer. Static\_cast and dynamic\_cast take this difference into account, but other cast operators do not take this into consideration and hence can give incorrect results. For example, in the use of multiple inheritance, the address of an object may be different than one of its base class sub-objects, causing the potential for the exploitable access of adjacent memory.

C++ permits the change of constant or volatile properties as part of a conversion. Such conversions, unless done in extremely limited ways, puts the program at risk of creating undefined behavior.

A typical use of pointer conversion in C++ is where there is a hierarchy of classes declared, as in:

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

 struct Derived: Base { };

Where a Base pointer needs to be converted to Derived pointer, dynamic\_cast will check at runtime that the pointer is to an object of the correct type. If it’s not, either nullptr will be returned, or an error exception thrown.

Pointer casts to a more strictly aligned pointer type is undefined behaviour.

Reinterpret\_cast for pointer-interconvertible on objects (see clause 6.9.2 of IS 14882)

C++ permits reinterpret\_cast to be used to convert a pointer to an object, a, to a pointer to another object, b, only in specific restricted circumstances, i.e., when

* a and b are the same object,
* either a or b is a standard-layout union object and the other is a non-static data member of that object,
	+ Examples:
	union A { int i; double d; } a;
	int\* iptr = reinterpret\_cast<int\*>(&a);
	double\* dptr = reinterpret\_cast<double\*>(&a);
	A\* uptr1 = reinterpret\_cast<A\*>(iptr);
	A\* uptr2 = reinterpret\_cast<A\*>(dptr);
* either a or b is a standard-layout class object and the other is the first non-static data member of that object,
	+ Examples:
	struct B { int i; double d; } b;
	int\* iptr = reinterpret\_cast<int\*>(&b);
	B\* bptr = reinterpret\_cast<B\*>(iptr);
* either a or b is a standard-layout class object with no non-static data members and the other is the first base class subobject of that object, or,
	+ Examples:
	struct A { double d; };
	struct B : A { static int i; } b;
	double\* dptr = reinterpret\_cast<double\*>(&b.d);
	B\* cptr = reinterpret\_cast<B\*>(dptr);
* there exists an object c where a and c are pointer-interconvertible and c and b are pointer-interconvertible.

In essence, such pointer-interconvertibility implies objects a and b have the same address, however, having the same address does not imply a and b are pointer-interconvertible! For example, an array and its first element have the same address but they are not pointer-interconvertible. This means that one cannot use reinterpret\_cast to cast an array object to the type of its first element or vice versa. [Reference: ISO 14882 Section 6.9.2 [basic.compound], Paragraph 4].

### 6.11.2 Guidance to language users

* Follow the advice provided by TR 24772-1 clause 6.11.5.
* Avoid the C-style cast, reinterpret\_cast, and casts to and from void\*.
* For conversions that remove the constant qualification, see the guidance in TR24772-1 clause 8.2.5
* or volatile qualifications
* When downcasting, prefer dynamic\_cast and explicitly handle the possible failure cases.
* References???
* Heed compiler warnings that are issued for pointer conversion instances. The decision may be made to avoid all conversions so any warnings must be addressed. Note that casting into and out of void \* pointers will most likely not generate a compiler warning as this is valid in C++
* Use new and delete to allocate/deallocate memory, rather than malloc/free

## 6.12 Pointer Arithmetic [RVG]

### 6.12.1 Applicability to language

The vulnerabilites as described in TR 24772-1 clause 6.12.1 also apply to C++ pointers. Analogous vulnerabilities also apply to C++ iterators.

Although based on the same implementation principles, iterators provide a layer of abstraction over pointer arithmetic. Their use typically restricts the arithmetic to the safe access to elements of the container. This restriction is enforced by the typical usage, not necessarily by the capability of iterators.

### 6.12.2 Guidance to language users

This subclause requires a complete rewrite.

* Follow the guidance of clause 6.8.2.
* Use iterators in lieu of pointers and pointer arithmetic. <<<John McF. to provide list of extras.>>>
* Use an iterator that checks against the bounds of the container before performing the intended operation on the container.
* Consider an outright ban on pointer arithmetic due to the error-prone nature of pointer arithmetic.
* Verify that all pointers are assigned a valid memory address for use.

## 6.13 NULL Pointer Dereference [XYH]

### 6.13.1 Applicability to language

The vulnerability as described in TR 24772-1 clause 6.13 exists in C++,…

C++ provides a number of mechanisms that allow the programmer to create, manipulate and destroy objects without the explicit use of raw pointers.

1. Containers manage memory and separate memory management from the use of objects.
2. The container interface throws an exception if any container cannot be allocated.
3. Smart pointer creation functions allocate heap memory and handle memory management.
4. References provide similar functionality as pointers, but cannot be null.

C++ mechanisms new, by default, throws an exception if the allocated object cannot be created (i.e. if a null pointer would be returned). C++ does provide other allocation mechanism, including C malloc and a non-throwing new, that are not recommended for general use.

See C++ Core Guidelines R: Resource Management, and CERT EXP34-C “Do not dereference null pointers”

### 6.13.2 Guidance to language users

When dereferencing objects of pointer-like types that may contain a null value, follow the guidance from TR 24772-3 clause 6.13.2.

* Avoid the use of direct memory allocation. Prefer the use of library facilities such as std::make\_unique, and std::make\_shared.
* Consider using std::array when the size of the array is known at compile time.
* Consider using std::vector instead of dynamic memory allocation of an array of dynamic size.
* Use references to reduce the number of places where pointers are dereferenced.
* Do not suppress exceptions on memory allocation. If exceptions are suppressed, follow the guidance of TR 24772-3 clause 6.13.2.

## 6.14 Dangling Reference to Heap [XYK]

### 6.14.1 Applicability to language

This subclause requires a complete rewrite.

C allows memory to be dynamically allocated primarily through the use of of malloc(), calloc(), and realloc(). C allows a considerable amount of freedom in accessing the dynamic memory. Pointers to the dynamic memory can be created to perform operations on the memory. Once the memory is no longer needed, it can be released through the use of free(). However, freeing the memory does not prevent the use of the pointers to the memory and issues can arise if operations are performed after memory has been freed.

Consider the following segment of code:

 int foo() {

 int \*ptr = malloc (100\*sizeof(int));// allocate space for 100 integers

 if (ptr != NULL) { // check that the memory could be allocated

 // perform operations on the dynamic memory

 free (ptr); // memory no longer needed, so free it

 // program performing other operations

 ptr[0] = 10; // ERROR – memory being used after released

 …

 }

 }

The use of memory in C after it has been freed is undefined. Depending on the execution path taken in the program, freed memory may still be free or may have been allocated via another malloc()or other dynamic memory allocation. If the memory that is used is still free, use of the memory may be unnoticed. However, if the memory has been reallocated, altering of the data contained in the memory can result in data corruption. Determining that a dangling memory reference is the cause of a problem and locating it can be difficult.

Setting and using another pointer to the same section of dynamically allocated memory can also lead to undefined behaviour. Consider the following section of code:

 int foo() {

 int \*ptr = malloc (100\*sizeof(int));/\* allocate space for 100 integers \*/

 if (ptr != NULL) { /\* check to see that the memory

 could be allocated \*/

 int ptr2 = &ptr[10]; /\* set ptr2 to point to the 10th

 element of the allocated memory \*/

 … /\* perform some operations on the

 dynamic memory \*/

 free (ptr); /\* memory is no longer needed \*/

 ptr = NULL; /\* set ptr to NULL to prevent ptr

 from being used again \*/

 … /\* program continues performing

 other operations \*/

 ptr2[0] = 10; /\* ERROR – memory is being used

 after it has been released via ptr2 \*/

 …

 }

 return (0);

 }

Dynamic memory was allocated via a malloc()and then later in the code, ptr2 was used to point to an address in the dynamically allocated memory. After the memory was freed using free(ptr) and the good practice of setting ptr to NULL was followed to avoid a dangling reference by ptr later in the code, a dangling reference still existed using ptr2.

### 6.14.2 Guidance to language users

This subclause requires a complete rewrite.

* Follow the advice provided by TR 24772-1 clause 6.15.2.

Set a freed pointer to NULL immediately after a free()call, as illustrated in the following code:

free (ptr);

ptr = NULL;

* Do not create and use additional pointers to dynamically allocated memory.
* Only reference dynamically allocated memory using the pointer that was used to allocate the memory.

## 6.15 Arithmetic Wrap-around Error [FIF]

### 6.15.1 Applicability to language

This subclause requires a complete rewrite.

Given the fixed size of integer data types, continuously adding one to an *unsigned* integer eventually will cause the value to go from the maximum possible value to a small value. C permits this to happen without any detection or notification mechanism. Continuously adding one to a *signed* integer eventually will cause undefined behaviour.

For example, consider the following code for a short int containing 16 bits:

 int foo( short int i ) {

 i++;

   return i;

 }

Calling foo with the value of 32767 would cause undefined behaviour, such as wrapping to -32768, or trapping. Manipulating a value in this way can result in unexpected results such as overflowing a buffer.

C is often used for bit manipulation. Part of this is due to the capabilities in C to mask bits and shift them. Another part is due to the relative closeness C has to assembly instructions. Manipulating bits on a signed value can inadvertently change the sign bit resulting in a number potentially going from a positive value to a negative value.

In C, bit shifting by a value that is greater than the size of the data type or by a negative number is undefined. The following code, where a int is 16 bits, would be undefined when j >= 16 or j is negative:

 int foo( int i, const int j ) {

   return i>>j;

 }

### 6.15.2 Guidance to language users

This subclause requires a complete rewrite.

* Be aware that any of the following operators have the potential to wrap in C:

a + b a – b a \* b a++ a--

a += b a -= b a \*= b a << b a >> b -a

* Use defensive programming techniques to check whether an operation will overflow or underflow the receiving data type. These techniques can be omitted if it can be shown at compile time that overflow or underflow is not possible.
* Only conduct bit manipulations on unsigned data types. The number of bits to be shifted by a shift operator should lie between 1 and (n-1), where n is the size of the data type.

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

### 6.16.1 Applicability to language

This subclause requires a complete rewrite.

The issues for C are well defined in TR 24772-1 clause 6.16 *Using Shift Operations for Multiplication and Division [PIK].* Also see clause *6.15 Arithmetic Wrap-around Error [FIF]*.

### 6.16.2 Guidance to language users

This subclause requires a complete rewrite.

The guidance for C users is well defined in TR 24772-1 clause 6.16 *Using Shift Operations for Multiplication and Division [PIK].* Also see, *6.15 Arithmetic Wrap-around Error [FIF].*

## 6.17 Choice of Clear Names [NAI]

### 6.17.1 Applicability to language

This subclause requires a complete rewrite to have it reflect C++ issues..

C is somewhat susceptible to errors resulting from the use of similarly appearing names. C does require the declaration of variables before they are used. However, C allows scoping so that a variable that is not declared locally may be resolved to some outer block and a human reviewer may not notice that resolution. Variable name length is implementation specific and so one implementation may resolve names to one length whereas another implementation may resolve names to another length resulting in unintended behaviour.

As with the general case, calls to the wrong subprogram or references to the wrong data element (when missed by human review) can result in unintended behaviour.

### 6.17.2 Guidance to language users

This subclause requires a complete rewrite.

* Use names that are clear and non-confusing.
* Use consistency in choosing names.
* Keep names short and concise in order to make the code easier to understand.
* Choose names that are rich in meaning.
* Keep in mind that code will be reused and combined in ways that the original developers never imagined.
* Make names distinguishable within the first few characters due to scoping in C. This will also assist in averting problems with compilers resolving to a shorter name than was intended.
* Do not differentiate names through only a mixture of case or the presence/absence of an underscore character.
* Avoid differentiating through characters that are commonly confused visually such as ‘O’ and ‘0’, ‘l’ (lower case ‘L’), ‘I’ (capital ‘I’) and ‘1’, ‘S’ and ‘5’, ‘Z’ and ‘2’, and ‘n’ and ‘h’.
* Develop coding guidelines to define a common coding style and to avoid the above dangerous practices.

## 6.18 Dead Store [WXQ]

### 6.18.1 Applicability to language

The vulnerability as documented in TR 24772-1 clause 6.18 exists in C++.

Issue of finalization of class objects

For Volatile, what do you do to ensure that a write reaches memory?

### 6.18.2 Guidance to language users

* Use compilers and static analysis tools to identify dead stores in the program.
* If variables are intended to be accessed by other execution threads, mark them as atomic.
* If variables are intended to be accessed by external devices, mark them as volatile.
* Declare variables as volatile when they are intentional targets of a store whose value does not appear to be used.

## 6.19 Unused Variable [YZS]

### 6.19.1 Applicability to language

The vulnerability as documented in TR 24772-1 clause 6.19 exists in C++.

### 6.19.2 Guidance to language users

* Follow the guidance of TR 24772-1 clause 6.19.5.
* Resolve all compiler warnings for unused variables.

## 6.20 Identifier Name Reuse [YOW]

### 6.20.1 Applicability to language

This subclause requires a complete rewrite to have it reflect C++ issues.

C allows scoping so that a variable that is not declared locally may be resolved to some outer block and that resolution may cause the variable to operate on an entity other than the one intended.

Because the variable name var1 was reused in the following example, the printed value of var1 may be unexpected.

int var1; /\* declaration in outer scope \*/

var1 = 10;

{

 int var2;

 int var1; /\* declaration in nested (inner) scope \*/

 var2 = 5;

 var1 = 1; /\* var1 in inner scope is 1 \*/

}

 print (“var1=%d\n”, var1); /\* will print “var1=10” as var1 refers \*/

 /\* to var1 in the outer scope \*/

Removing the declaration of var2 will result in a diagnostic message being generated making the programmer aware of an undeclared variable. However, removing the declaration of var1 in the inner block will not result in a diagnostic as var1 will be resolved to the declaration in the outer block and a programmer maintaining the code could very easily miss this subtlety. The removing of inner block var1 will result in the printing of var1=1 instead of var1=10.

### 6.20.2 Guidance to language users

This subclause requires a complete rewrite.

* Ensure that a definition of an entity does not occur in a scope where a different entity with the same name is accessible and can be used in the same context. A language-specific project coding convention can be used to ensure that such errors are detectable with static analysis.
* Ensure that a definition of an entity does not occur in a scope where a different entity with the same name is accessible and has a type that permits it to occur in at least one context where the first entity can occur.
* Ensure that all identifiers differ within the number of characters considered to be significant by the implementations that are likely to be used, and document all assumptions.

## 6.21 Namespace Issues [BJL]

### 6.21.1 Applicability to language

This subclause requires a complete rewrite to have it reflect C++ issues.

## 6.22 Initialization of Variables [LAV]

### 6.22.1 Applicability to language

The vulnerability as described in TR 24772-1 exists in C++.

C++ provides language capabilities to mitigate the effects of uninitialized variables as follows:

See C++ Core Guidelines ES.20 and CERT C++ Coding Guidelines EXP53-CPP

Need a list of references TBD – (AI – J. Daniel Garcia)

Readers should 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 Guidance to language users

* Follow the guidance provided in C++ Core Guidelines, section Class hierarchies, and Expressions and Statements and SEI CERT C++ Coding Standard section EXP53-CPP (and possibly more).

## 6.23 Operator Precedence and Associativity [JCW]

### 6.23.1 Applicability to language

This subclause requires a complete rewrite to have it reflect C++ issues.

Operator precedence and associativity in C are clearly defined.

Mixed logical operators are allowed without parentheses.

### 6.23.2 Guidance to language users

This subclause requires a complete rewrite.

* Follow the guidance provided in TR 24772-1 clause 6.23.5
* Use parentheses any time arithmetic operators, logical operators, and shift operators are mixed in an expression.

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

### 6.24.1 Applicability to language

Clause needs a complete rewrite.

C allows expressions to have side effects. If two or more side effects modify the same expression as in:

 int v[10];

 int i;

 /\* … \*/

 i = v[i++];

the behaviour is undefined and this can lead to unexpected results. Either the “i++” is performed first or the assignment i=v[i] is performed first, or some other undefined behaviour occurs. Because the order of evaluation can have drastic effects on the functionality of the code, this can greatly impact portability.

There are several situations in C where the order of evaluation of subexpressions or the order in which side effects take place is unspecified including:

* The order in which the arguments to a function are evaluated (C, Section 6.5.2.2,"Function calls").
* The order of evaluation of the operands in an assignment statement (C, Section 6.5.16,"Assignment operators").
* The order in which any side effects occur among the initialization list expressions is unspecified. In particular, the evaluation order need not be the same as the order of subobject initialization (C, Section 6.7.9, “Initialization").

Because these are unspecified behaviours, testing may give the false impression that the code is working and portable, when it could just be that the values provided cause evaluations to be performed in a particular order that causes side effects to occur as expected.

### 6.24.2 Guidance to language users

* Follow the guidance provided in TR 24772-1 clause 6.24.5
* Expressions should be written so that the same effects will occur under any order of evaluation that the C standard permits since side effects can be dependent on an implementation specific order of evaluation.
* Become familiar with Annex C of the C standard ISO/IEC 9899:2011 [4], which is a list of the sequence points that enforce an ordering of computations.

## 6.25 Likely Incorrect Expression [KOA]

### 6.25.1 Applicability to language

This subclause requires a complete rewrite to have it reflect C++ issues.

C has several instances of operators which are similar in structure, but vastly different in meaning. This is so common that the C example of confusing the Boolean operator “==” with the assignment “=” is frequently cited as an example among programming languages. Using an expression that is technically correct, but which may just be a null statement can lead to unexpected results.

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

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 (perfectly valid in C) or whether the programmer made the common mistake of using an “=” instead of a “==”. In order to prevent this confusion, it is suggested that any assignments in contexts that are easily misunderstood be moved outside of the Boolean expression. 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.

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 this a null statement

 {

 /\* … \*/

 }

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.

### 6.25.2 Guidance to language users

* Simplify statements with interspersed comments to aid in accurately programming functionality and help future maintainers understand the intent and nuances of the code. The flexibility of C permits a programmer to create extremely complex expressions.
* Avoid assignments embedded within other statements, as these can be problematic. Each of the following would be clearer and have less potential for problems if the embedded assignments were conducted outside of the expressions:

 int a,b,c,d;

 /\* … \*/

 if ((a == b) || (c = (d-1))) /\* the assignment to c may not

 occur if a is equal to b \*/

 or:

 int a,b,c;

 /\* … \*/

 foo (a=b, c);

 Each is a valid C statement, but each may have unintended results.

* Give null statements a source line of their own. This, combined with enforcement by static analysis, would make clearer the intention that the statement was meant to be a null statement.
* Consider the adoption of a coding standard that limits the use of the assignment statement within an expression.

## 6.26 Dead and Deactivated Code [XYQ]

### 6.26.1 Applicability to language

The vulnerability as documented in TR 24772-1 clause 6.26 exists in C++.

### 6.26.2 Guidance to language users

* Follow the guidance of TR 24772-1 clause 6.26.5.

## 6.27 Switch Statements and Static Analysis [CLL]

### 6.27.1 Applicability to language

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 Guidance to language users

* Apply the guidance provided in TR 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:

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

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++);

 {

 a = a + b[i];

 }

 int c = 0;

 int x = 0;

 for (int i=0; i<10; i++)

 c = c + b[i];

 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:

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 Guidance to language users

* Follow the rules provided in TR 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.

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

C++ allows the modification of loop control variables within a loop. Though 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 valid:

 int a;

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

 …

 if (a > 7)

 i = 10;

 …

 }

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

C++ also permits the use of multiple variable of the same type in the loop header

Mitigation – range for statement – document with an example (see ES.71) – Gabriel

### 6.29.2 Guidance to language users

* Apply the guidance of TR 24772-1 clause 6.29.5.
* Do not modify a loop control variable within a loop. Even though the capability exists in C, it is still considered to be a poor programming practice.
* Use a range for loop in preference to general loops
* Alternatively, use std library functions copy, accumulate, transform, for\_each, etc. in preference to general loops.
* Something about multiple loop control variables in the same loop?

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

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

### 6.30.1 Applicability to language

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 TR 24772-1 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 Guidance to language users

* Follow the guidance of TR 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

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. Also, C 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.

### 6.31.2 Guidance to language users

* Write clear and concise structured code to make code as understandable as possible.
* Avoid the use of longjmp
* Avoid the use of goto except in the case of exiting a nested loop.

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

C++ provides both *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.

An object can be modified in a function by passing the address to the object to the function, for example

 void swap(int \*x, int \*y) { // C-style

 int t = \*x;

 \*x = \*y;

 \*y = t;

 }

A call to this function is swap( &a, &b);

In a preferred style (below), an object may be passed to a function by reference, which eliminates many of the problems enumerated in TR 24772-1 clause 6.32.1 and 6.32.3.

void swap(int & x, int & y) { // C++-style which is like std::swap

 int t = x;

 x = y;

 y = t;

 }

This function is called by swap(a,b);

Where x and y are integer pointer formal parameters, and \*x and \*y in the swap()function body dereference the pointers to access the integers.

C 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.

*Paragraph about the violation of the keyword “restrict” in Part 3. – C++ does not have this keyword. Think about the issue.*

### 6.32.2 Guidance to language users

* Follow the advice of TR 24772-1 clause 6.32.5.
* Use caution for reevaluation of function calls in parameters with macros.
* Use caution when passing the address of an object. The object passed could be an alias[[2]](#footnote-2). Aliases can be avoided by following the respective guidelines of TR 24772-1 Clause 6.32.5.

See also the C++ Core Guidelines F.7 through F.48.

## 6.33 Dangling References to Stack Frames [DCM]

### 6.33.1 Applicability to language

C++ allows one variable to refer to another variable. For example, a pointer variable can contain the address of another variable; a reference can be bound to a variable; and an iterator can point to a portion of a variable (in this case a container). Should the referencing variable outlive the referenced variable, the subsequent operations through the referencing variable will have undefined behavior.

For example

int \*bad\_pointer() {
  int a = 0;
  return &a;
 }

int& bad\_reference() {
  int a = 0;
  return a;
 }

std::array<int,3>::iterator bad\_iterator()
 {
  std::array<int,3> a = { 1, 2, 3 };
  return a.begin();
 }

auto bad\_lambda() {

    int x = 0;

    return [&] { x = 1; };

}

void erroneous\_use() {
  std::cout << \*bad\_pointer();
  std::cout << bad\_reference();
  std::cout << \*bad\_iterator();

  std::cout << bad\_lambda()();
 }

### 6.33.2 Guidance to language users

* Do not assign the address of an object, or reference to any entity where the referencing entity persists after the object has ceased to exist. This is done in order to avoid the possibility of a dangling reference.
* Do not return the address of a local variable as the result of a function call.
* Do not return a local variable as the result of a function returning a reference type
* Avoid capturing by reference in lambdas that will be used non-locally, including return, or passing it to another thread, or stored in dynamic memory

See also C++ Core Guidelines F.53, …

## 6.34 Subprogram Signature Mismatch [OTR]

### 6.34.1 Applicability to language

In general, there must be a match between the number of parameters in a function call and the number of arguments in the function definition, with the exception of va\_arg functions f(…).

C++ allows a variable number of arguments in function calls. A good example of a va\_arg function the printf() function. This is specified in the function call by terminating the list of parameters with an ellipsis (, ...). After the comma, no information about the number or types of the parameters is supplied. The use of this feature outside of special situations can be the basis for vulnerabilities.

### 6.34.2 Guidance to language users

* Follow the guidelines of TR 24772-1 clause 6.34.5.
* Avoid va\_arg functions .

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 6.35.

### 6.35.2 Guidance to language users

* Apply the guidance described in TR 24772-1 clause 6.35.5.

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

### 6.36.1 Applicability to language

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. For example

 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.

[[nodiscard]] errnum foo( int a, int b);

. . .

foo(x, y); // compiler error.

if( auto e = foo(a,b); e == 0) { // no compiler error

// success

}

else {

// handle errors

}

 *Should we include a discussion about error\_code??? AI – Michael Wong*

Discuss global error states, such as errno (which is thread-local) but still static.

Global state for error codes is hard to manage and it is easy to forget to check it (C++ Core Guidelines E.28).

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.

### 6.36.2 Guidance to language users

* Follow the mitigation mechanisms of subclause 6.36.5 of TR 24772-1.
* Check the returned error status upon return from a function. The C standard library functions provide an error status as the return value and sometimes in an additional global error value.
* Use static analysis tools to detect and report missing or ineffective error detection or handling.
* Avoid error handling based on global state.
* Use [[nodiscard]] to prevent callers from ignoring error values.
* Prefer throwing exceptions to returning error values.
* Handle an error as close as possible to the origin of the error but as far out as necessary to be able to deal with the error.
* Use destructors to manage the finalization of the current context upon exit, whether erroneous or not.

See also C++ Core Guidelines E.1, E.2, E.5, E.6, E.13, E.17, E.19, E.25, and E.28.

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

### 6.37.1 Applicability to language

This subclause requires a complete rewrite to have it reflect C++ issues.

The primary way in C that a reinterpretation of data is accomplished is through a union which may be used to interpret the same piece of memory in multiple ways. If the use of the union members is not managed carefully, then unexpected and erroneous results may occur.

C allows the use of pointers to memory so that an integer pointer could be used to manipulate character data. This could lead to a mistake in the logic that is used to interpret the data leading to unexpected and erroneous results.

*Wait for Gabriel to help analyze this*

### 6.37.2 Guidance to language users

* Follow the guidelines of TR 24772-1 clause 6.38.5.
* When using unions, implement an explicit discriminant and check its value before accessing the data in the union.

## 6.38 Deep vs. Shallow Copying [YAN]

### 6.38.1 Applicability to Language

This vulnerability only arises in C++ when there is a mismatch between the object’s copy semantics and the programmer’s intent. (references to Core Guidelines C.22)

C++ objects, by default, are copied member-wise. Each class type may define its own copy, move and assignment operations, allowing a class author to choose an appropriate depth for these operations. Class member types should be chosen to have copy and move semantics that support the semantics of the enclosing class.

<This may belong elsewhere – TBD> C++ provides the “string view” mechanism as safer pointers to strings. Updates through string view are prohibited, but the initial non “view” value can be updated and this change will be seen by all viewers, even if they are dependent on fixed value.

Note: in C++, this is more commonly known as member-wise copying vs semantic copying, or owning vs observing rights.

Note: Why CERT does not address this issue – involves programmer intent and not readily tool-checkable.

### 6.38.2 Guidance to language users

* Prefer the composition of most types from types that have either value semantics or semantics that support the intended copy and move semantics of the enclosing type.
* When the above is not achievable, ensure that the copy assignment operator, copy constructor, move assignment operator, move constructor and destructor provide the desired semantics.
* Avoid the use of raw pointers with the copy operation and (finish or delete)
* Follow the guidance of C++ core guidelines C.20, C.22, C.32, C.67
* *<This may belong elsewhere – TBD>* Avoid updating the value of a string while there are valid string views in existence.

## 6.39 Memory Leak and Heap Fragmentation [XYL]

### 6.39.1 Applicability to language

C++ uses destructors, and a pattern called Resource Acquisition Is Initialization (RAII) which performs recovery of resources. Destructors (and therefore memory and resource releases) are deterministically ordered with respect to other events on their thread. Object destructors will not be called

* When an unhandled exception escapes its thread of execution
* Under conditions of abnormal termination

See CERT ERR50-CPP for list of cases.

The memory leak vulnerability documented in TR24772-1 clause 6.39 exists in C++, unless the programmer takes steps to avoid it. The steps mentioned above will mitigate most memory leak issues.

The mechanisms std::shared\_ptr and std::shared\_future and similarly constructed reference-counting user code do not detect cycles which will cause leaks because the shared pointers (and hence what they point to) will not be destroyed.

### 6.39.2 Guidance to language users

* Use containers and smart pointers in preference to direct (manual) memory management.
* Follow C++ Core guidelines section R and CERT MEM51.
* For heap fragmentation issues, follow the guidance of TR 24772-1 clause 6.39.5. In particular, create pools of fixed size with user-defined operators new and operators delete.
* Use dynamic analysis tools to detect cycles.
* Break cycles, for example by using std::weak\_ptr or appropriate weak pointers.
* Use std::abort() or std::terminate() and related functions only in extreme situations. See CERT ERR50-CPP for list of cases.
* Use debugging tools such as leak detectors to help identify unreachable memory.

## 6.40 Templates and Generics [SYM]

### 6.40.1 Applicability to language

*The following text came from Part one. Consider its relevance for C++.*

The value of generics comes from having a single piece of code that supports some behaviour in a type independent manner. This simplifies development and maintenance of the code. It should also assist in the understanding of the code during review and maintenance, by providing the same behaviour for all types with which it is instantiated.

Problems arise when the use of a generic actually makes the code harder to understand during review and maintenance, by not providing consistent behaviour.

In most cases, the generic definition will have to make assumptions about the types it can legally be instantiated with. For example, a sort function requires that the elements to be sorted can be copied and compared. If these assumptions are not met, the result is likely to be a compiler error. For example if the sort function is instantiated with a user defined type that does not have a relational operator. Where ‘misuse’ of a generic leads to a compiler error, this can be regarded as a development issue, and not a software vulnerability.

Confusion, and hence potential vulnerability, can arise where the instantiated code is apparently invalid, but does not result in a compiler error. For example, a generic class defines a set of members, a subset of which rely on a particular property of the instantiation type (such as a generic container class with a sort member function, only the sort function relies on the instantiating type having a defined relational operator). In some languages, such as C++, if the generic is instantiated with a type that does not meet all the requirements but the program never subsequently makes use of the subset of members that rely on the property of the instantiating type, the code will compile and execute (for example, the generic container is instantiated with a user defined class that does not define a relational operator, but the program never calls the sort member of this instantiation). When the code is reviewed the generic class will appear to reference a member of the instantiating type that does not exist.

*The problem as described in the two prior paragraphs can be reduced by a language feature (such as the concepts language feature being designed by the C++ committee).* (RESEARCH – AI Clive.).

Similar confusion can arise if the language permits specific methods of an instance of a generic to be explicitly defined, rather than using the common code, so that behaviour is not consistent for all instantiations. For example, for the same generic container class, the sort member normally sorts the elements of the container into ascending order. In some languages, a ‘special case’ can be created for the instantiation of the generic with a particular type. For example, the sort member for a ‘float’ container may be explicitly defined to provide different behaviour, say sorting the elements into descending order. Specialization that does not affect the apparent behaviour of the instantiation is not an issue.

(C++-specific text, move when appropriate – AI Clive.).*Again, for C++, there are some irregularities in the semantics of arrays and pointers that can lead to the generic having different behaviour for different, but apparently very similar, types. In such cases, specialization can be used to enforce consistent behaviour.*

*Again, for C++, there are some irregularities in the semantics of arrays and pointers that can lead to the generic having different behaviour for different, but apparently very similar, types. In such cases, specialization can be used to enforce consistent behaviour.*

This subclause requires a complete rewrite to have it reflect C++ issues.

### 6.40.2 Guidance to language users

## 6.41 Inheritance [RIP]

## 6.41.1 Applicability to language

Inheritance, the ability to create enhanced and/or restricted object classes based on existing object classes, can introduce a number of vulnerabilities, both inadvertent and malicious. Because inheritance allows the overriding of methods of the parent class and because object-oriented systems are designed to separate and encapsulate code and data, it can be difficult to determine where in the hierarchy an invoked method is actually defined.

Also, since an overriding method does not need to call the method in the parent class that has been overridden, essential manipulation of class data may be bypassed.

This can be especially dangerous in copy assignment operator and move assignment operators and in particular when private data components (that is, data components not visible to methods of subclasses) of the parent class are left unchanged. Serious violations of type invariants can arise as a consequence.

Multiple inheritance adds additional complexities to the resolution of method invocations.

The use of inheritance can lead to an exploitable application vulnerability or negatively impact system safety in several ways:

* Execution of malicious redefinitions, which can occur through the insertion of a class into the class hierarchy that overrides commonly called methods in the parent classes.
	+ mitigation – make member functions ‘final’,
	+ reduce the use of inheritance
* Accidental override, where a member function is defined that inadvertently overrides a member function that has already been defined in a parent class.
	+ Mitigation – use “override” and “final” keywords on member functions to generate compiler diagnostics when overriding is accidental
* Accidental failure to override, when a method is incorrectly named or the parameters are not defined properly, and thus does not override a member function in a parent class.
	+ Mitigation – use “override” and “final” keywords on member functions to generate compiler diagnostics when overriding is accidental
* Breaking of class invariants, which can be caused by redefining methods that assign, move, or validate class data without including the assigning, moving or validating in the overriding member function. This applies particularly to class invariants involving data of the parent class not visible in methods of the subclass. Inherited methods of the parent that have access to these “private” components will likely fail, if the components are set inappropriately.
	+ Mitigation – if any class invariant depends upon a value of a data member, then make that member private
* Direct reading and writing of visible class members when matching getting and setting member functions include additional functionality.
	+ Guidance: make data members private and provide a public interface to access them that preserves class invariants.

These vulnerabilities can increase dramatically as the complexity of the hierarchy increases, especially in the use of multiple inheritance.

As member functions are inherited from multiple chains of ancestors, the determination of which member function implementations exist and are being called, becomes increasingly more difficult for the programmer. Understanding which member functions and data members apply to a given (sub)class becomes exceedingly difficult if these methods or components are inherited homographs (i.e., data components with identical names or member functions with identical signatures). Misunderstandings lead to inadvertent coding errors. The complexity increases even more when multiple inheritance is used to model “has-a“ relationships (see subclause [6.42 Violations of the Liskov substitution principle [BLP])](#_6.42_Violations_of_1): member functions never intended to be applicable to instances of a subclass are inherited nevertheless. For example, an instance of class aircraftCarrier may be “turn”ed merely because it obtained its propulsion screw by a “has-a“-inheritance with “turn” being an obviously meaningful method for the class of propulsionScrew. Meanwhile the user has a quite different expectation of what it means to turn an aircraft carrier. The complications increase if the carrier inherits twice from the class propulsionScrew because it has two propulsion screws.

Changes in the execution of methods can be introduced by adding an unrelated but homographic member function (with signatures involving implicitly convertible types) anywhere is the hierarchies of ancestor classes during maintenance of the code. Malicious implementations can thus be added with each release of an object-oriented library and affect the behaviour of previously verified code. (see subclause [6.42 Violations of the Liskov substitution principle [BLP])](#_6.42_Violations_of_1)

* Guidance: Keep inheritance hierarchies short
* Guidance: Qualify the program to invoke member functions in explicit parent classes.
* Mitigation: use the ‘= delete’ construct to prevent a member function from being called due to an inheritance.

## 6.41.2 Guidance to language users

* Follow the guidance of 24772-1 clause 41.5.
* Avoid the use of multiple inheritance whenever possible.
* Avoid access to data components when getting and setting functions are available for them.
* Keep inheritance hierarchies short and narrow
* Prefer non-virtual functions to virtual functions
* Use “override” and “final” keywords on member functions to generate compiler diagnostics when overriding is accidental
* Use the ‘= delete’ construct to prevent a member function from being called due to an inheritance.
* If any class invariant depends upon a value of a data member, then make that member private
* Make data members private and provide a public interface to access them that preserves class invariants
* Provide complete documentation of all encapsulated data, and how each method affects that data for each object in the hierarchy.
* Inherit only from trusted sources, and, whenever possible, check the version of the parent classes during compilation and/or initialization.
* Provide a member function that provides versioning information for each class.
* Prohibit the use of public inheritance for “has-a” relationships. Use composition instead for “has-a”-relationships.
* Delegate assigning and moving of the parent’s data components by calling the corresponding operation of the parent type. You must delegate in particular when the parent has data components not visible to methods of the subclass. Alternatively, prohibit assignment and motion for classes intended to be base types. *(clarify – this has 2 possible meanings)*
* Avoid the creation of base classes that are both virtual and non-virtual in the same hierarchy.

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

## 6.42.1 Applicability to language

This vulnerability applies to C++ . It 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;
         }
     // ...
 };

## 6.42.2 Guidance to language users

* Obey all preconditions and postconditions of each member function, whether they are specified in the language or not.
* Prohibit the strengthening of preconditions (specified or not) by overriding member functions.
* Prohibit the weakening of postconditions (specified or not) by overriding member functions.
* Prohibit the use of public inheritance for “has-a” relationships. Use composition for “has-a”-relationships instead.
* Use static analysis tools that identify misuse of inheritance in the contract model.
* 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.

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

In C++, the vulnerability exists 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"; g(); } //call to g() will dispatch,

 //showing the vulnerability
};

class B : public A {
public:
    void f() override { std::cout << "B::f()\n"; g(); }
    //void g() override { std::cout << "B::g()\n"; f(); }
    //void h() override { std::cout << "B::h()\n"; g(); }
};

int main() {
    B b;
    A \* pA = &b;
    pA->f();
    std::cout << "---\n";
    pA->g();
}

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

## 6.43.2 Guidance to language users

* 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]

The following is from Part 1:

Object-oriented languages allow polymorphic variables, in which values of different classes can be stored at different times. In most of these languages, variables are declared to be of some class, while the actual value may be of a more specialized subclass. Polymorphic variables go hand in hand with method selection at run time, when the method defined for the actual subclass of the receiving object or controlling argument is invoked. This approach is safe, as method implementation and actual type of the object match by construction. If, however, the language permits casting of the polymorphic reference to process the object as if it were of the class casted to, several vulnerabilities arise. We distinguish the following casts:

* *upcasts*, where the cast is to a superclass,
* *downcasts*, where the cast is to a subclass and a check is made that the object is indeed of the target class of the cast (or a subclass thereof),
* *unsafe casts*, where there is no assurance that the object is of the casted class.

Distinct vulnerabilities arise for each of these cast types:

Upcasts are needed so that redefined methods can call upon the corresponding method of the parent class to achieve the respective portion of the needed functionality and then complete it for the extensions added by the subclass. Without calling the parent’s implementation of a method in the redefined method, the private components of the parent class are inaccessible to the redefined method. Hence there is a risk that they are no longer consistent with the overall state of the object. Inversely, if the issue is avoided by inheriting rather than redefining the method for a subclass, there is the risk that the subclass-specific parts are inconsistent with the overall state of the object or even uninitialized.

Downcasts carry the risk that the object is not of the correct class. If checked by the language, as language-defined downcasts typically are, an exception will occur in this case.

Unsafe casts allow arbitrary breaches of safety and security. See subclause  [6.11 Pointer Casting and Pointer Type Changes](#_6.11_Pointer_type) [HFC].

Note that some languages also have implicit upcasts and downcasts as part of the language semantics. The same issues apply as for explicit casts.

Part 3

Objects left in an inconsistent state by means of an upcast and a subsequent legitimate method call of the parent class can be exploited to cause system malfunctions.

Exceptions raised by failing downcasts allow Denial-of-Service attacks. Typical scenarios include the addition of objects of some unexpected subclasses in generic containers.

Unsafe casts to classes with the needed components allow reading and modifying arbitrary memory areas. See subclause [6.11 Pointer Casting and Pointer Type Changes](#_6.11_Pointer_type_1) [HFC] for more details.

## 6.44.1 Applicability to language

This vulnerability applies to C++. In addition to the upcast and downcast issues addressed in TR 24772-1 clause 6.44, this clause also addresses crosscasting, which is unique(?) to C++.

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.
* 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](#_6.11_Pointer_type_1) [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 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.

## 6.44.2 Guidance to language users

* Follow the advice provided in TR 24772-1 clause 6.44.5.
* If an upcast is needed, prefer using implicit conversion, since an explicit upcast adds unnecessary complexity for the reader.
* If a downcast or a crosscast is needed, prefer using dynamic\_cast because 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.
* Do not attempt to navigate class hierarchies using C-style casts or reinterpret\_cast.
* For any class that implements a virtual member function, consider marking that member function final in the definition of that class.

NOTE: This forbids any derived class to redefine the implementation and thereby precludes ambiguity, regardless of whether a call is qualified or not.

NOTE: Making instead the class final contradicts C++ Core Guideline C.139, so is not recommended here.

* Consider declaring virtual methods with protected or private visibility to preclude code from outside of the class hierarchy calling any specific implementation directly.

NOTE: This assumes that within the class hierarchy any qualified call is intentional and is the pattern of a non-public virtual interface.

See also C++ Core Guidelines ES.48, ES.49, C.146, C.147, C.148 and C.153.

## 6.45 Extra Intrinsics [LRM]

This vulnerability does not apply to C++ for the following reasons:

* When adding intrinsics, implementors are required to follow a specific name pattern that users are not allowed to use in definitions. See C++ standard clause 5.10 [Lex.name].

## 6.46 Argument Passing to Library Functions [TRJ]

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.

###  Applicability to language

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. 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 Guidance to language users

* Follow the guidelines of TR 24772-1 clause 6.47.5.
* Use translation modes provided 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]

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]](#footnote-3), leaving the developer the task of channeling everything through this common language system.

### 6.47.2 Guidance to language users

* Follow the guidelines of TR 24772-1 clause 6.48.5.

Next Items from TR 24772-1 clause 6.47.5:

* Use the inter-language methods and syntax specified by the applicable language standard(s)[[4]](#footnote-4).
* Understand the calling conventions of all languages used.
* For items comprising the inter-language interface:
* Understand the data layout of all data types used.
* Understand the return conventions of all languages used.
* Ensure that the language in which error check occurs is the one that handles the error.
* Avoid assuming that the language makes a distinction between upper case and lower case letters in identifiers.
* Avoid using a special character as the first character in identifiers.
* Avoid using long identifier names.

From Part 3 (C document):

* Minimize the use of those issues known to be error-prone when interfacing from C, such as
* passing character strings,
* dimension, bounds and layout issues of arrays,
* interfacing with other parameter formats such as call by reference or name,
* receiving return codes, and
* bit representation.
* Follow the guidance contained in Tr 24772-1 clause 6.47.5
* Use standard layout types for the interoperable interfaces.
* Use language linkage facilities that support the languages being used
* 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 – group – add the guidance from 6.47 Interoperability into the Core Guidelines.

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

Code that is dynamically linked may be different from the code that was tested. This may be the result of replacing a library with another of the same name or by altering an environment variable such as LD\_LIBRARY\_PATH on UNIX platforms so that a different directory is searched for the library file. Executing code that is different than that which was tested may lead to unanticipated errors or intentional malicious activity.

On some platforms, and in some languages, instructions can modify other instructions in the code space. Historically self-modifying code was needed for software that was required to run on a platform with very limited memory. It is now primarily used (or misused) to hide functionality of software and make it more difficult to reverse engineer or for specialty applications such as graphics where the algorithm is tuned at runtime to give better performance. Self-modifying code can be difficult to write correctly and even more difficult to test and maintain correctly leading to unanticipated errors.

Mechanism of failure:

Through the alteration of a library file or environment variable, the code that is dynamically linked may be different from the code which was tested resulting in different functionality.

On some platforms, a pointer-to-data can erroneously be given an address value that designates a location in the instruction space. If subsequently a modification is made through that pointer, then an unanticipated behaviour can result

### 6.48.1 Applicability to language

This subclause requires a complete rewrite to have it reflect C++ issues.

Most loaders allow dynamically linked libraries also known as shared libraries. Code is designed and tested using a suite of shared libraries which are loaded at execution time. The process of linking and loading is outside the scope of the C standard.

C can allow self-modifying code. In C there isn’t a distinction between data space and code space, executable commands can be altered as desired during the execution of the program. Although self-modifying code may be easy to do in C, it can be difficult to understand, test and fix leading to potential vulnerabilities in the code.

Self-modifying code can be done intentionally in C to obfuscate the effect of a program or in some special situations to increase performance. Because of the ease with which executable code can be modified in C, accidental (or maliciously intentional) modification of C code can occur if pointers are misdirected to modify code space instead of data space or code is executed in data space. Accidental modification usually leads to a program crash. Intentional modification can also lead to a program crash, but used in conjunction with other vulnerabilities can lead to more serious problems that affect the entire host.

### Guidance to language users

From Part 1:

* Verify that the dynamically linked or shared code being used is the same as that which was tested.
* Retest the application before use when it is possible that the dynamically linked or shared code has changed.
* Do not write self-modifying code except in extremely rare instances. Most software applications should never have a requirement for self-modifying code.
* In those extremely rare instances where its use is justified, limit the amount of self-modifying code and heavily document them.

From Part 3:

* Do not use self-modifying code except in rare instances. In those rare instances, self-modifying code in C can and should be constrained to a particular section of the code and well commented. In those extremely rare instances where its use is justified, limit the amount of self-modifying code and heavily document it.
* Verify that the dynamically linked or shared code being used is the same as that which was tested.
* Retest when it is possible that the dynamically linked or shared code has changed before using the application.

## 6.49 Library Signature [NSQ]

### 6.49.1 Applicability to language

This subclause requires a complete rewrite to have it reflect C++ issues.

From Part 1: 6.49.1

Programs written in modern languages may use libraries written in other languages than the program implementation language. If the library is large, the effort of adding signatures for all of the functions use by hand may be tedious and error-prone. Portable cross-language signatures will require detailed understanding of both languages, which a programmer may lack.

Integrating two or more programming languages into a single executable relies upon knowing how to interface the function calls, argument list and global data structures so the symbols match in the object code during linking.

Byte alignment can be a source of data corruption if memory boundaries between the programming languages are different. Each language may also align structure data differently.

6.49.3

When the library and the application in which it is to be used are written in different languages, the specification of signatures is complicated by inter-language issues.

As used in this vulnerability description, the term library includes the interface to the operating system, which may be specified only for the language used to code the operating system itself. In this case, any program written in any other language faces the inter-language interoperability issue of creating a fully-functional signature.

When the application language and the library language are different, then the ability to specify signatures according to either standard may not exist, or be very difficult. Thus, a translator-by-translator solution may be needed, which maximizes the probability of incorrect signatures (since the solution must be recreated for each translator pair). Incorrect signatures may or may not be caught during the linking phase.

Integrating C and another language into a single executable relies on knowledge of how to interface the function calls, argument lists and data structures so that symbols match in the object code during linking. Byte alignments can be a source of data corruption.

For instance, when calling Fortran from C, several issues arise. Neither C nor Fortran check for mismatch argument types or even the number of arguments. C passes arguments by value and Fortran passes arguments by reference, so addresses must be passed to Fortran rather than values in the argument list. Multidimensional arrays in C are stored in row major order, whereas Fortran stores them in column major order. Strings in C are terminated by a null character, whereas Fortran uses the declared length of a string. These are just some of the issues that arise when calling Fortran programs from C. Each language has its differences with C, so different issues arise with each interface.

Writing a library wrapper is the traditional way of interfacing with code from another language. However, this can be quite tedious and error-prone.

### 6.49.2 Guidance to language users

From Part 1, 6.49.5

* Use signatures to verify that the shared libraries used are identical to the libraries with which the code was tested.
* Use a tool, if possible, to automatically create the interface wrappers.

##

## Unanticipated Exceptions from Library Routines [HJW]

This subclause requires a complete rewrite to have it reflect C++ issues.

From Part 1: 6.50.1

A library in this context is taken to mean a set of software routines produced outside the control of the main application developer, usually by a third party, and where the application developer may not have access to the source. In such circumstances, the application developer has limited knowledge of the library functions, other than from their behavioural interface.

Whilst the use of libraries can present a number of vulnerabilities, the focus of this vulnerability is any undesirable behaviour that a library routine may exhibit, in particular the generation of unexpected exceptions.

From Part 1: 6.50.3

In some languages, unhandled exceptions lead to implementation-defined behaviour. This can include immediate termination, without for example, releasing previously allocated resources. If a library routine raises an unanticipated exception, this undesirable behaviour may result.

It should be noted that the considerations of subclause [6.36 Ignored Error Status and Unhandled Exceptions [OYB]](#_6.36_Ignored_error), are also relevant here.

### 6.50.1 Applicability to language

### 6.50.2 Guidance to language users

From 6.50.5

* Wrap all library calls within a ‘catch-all’ exception handler (if the language supports such a construct), so that any unanticipated exceptions can be caught and handled appropriately. This wrapping may be done for each library function call or for the entire behaviour of the program, for example, having the exception handler in main for C++. However, note that the latter is not a complete solution, as static objects are constructed before main is entered and are destroyed after it has been exited. Consequently, MISRA C++ [36] bars class constructors and destructors from throwing exceptions (unless handled locally).
* Alternatively, use only library routines for which all possible exceptions are specified.

## 6.51 Pre-processor Directives [NMP]

### 6.51.1 Applicability to language

This subclause requires a complete rewrite to have it reflect C++ issues.

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 if the arguments have side effects or are pre-processor directives as described by C §6.10 [1]. Additionally, the arguments and body of function-like macros should be fully parenthesized to avoid unintended and undefined behaviour [2].

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) [2]:

#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));

this is 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 [2]:

#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 Guidance to language users

* Replace macro-like functions with inline functions where possible. Although making a function inline only suggests to the compiler that the calls to the function be as fast as possible, the extent to which this is done is implementation-defined. Inline functions do offer consistent semantics and allow for better analysis by static analysis tools.
* Ensure that if a function-like macro must be used, that its arguments and body are parenthesized.
* Do not embed pre-processor directives or side-effects such as an assignment, increment/decrement, volatile access, or function call in a function-like macro.

## 6.52 Suppression of Language-defined Run-time Checking [MXB]

This subclause requires a complete rewrite to have it reflect C++ issues.

##

## 6.53 Provision of Inherently Unsafe Operations [SKL]

### 6.53.1 Applicability to language

This subclause requires a complete rewrite to have it reflect C++ issues.

### 6.53.2 Guidance to language users

* Follow the guidelines of TR 24772-1 clause 6.54.5.

## 6.54 Obscure Language Features [BRS]

### 6.54.1 Applicability of language

This subclause requires a complete rewrite to have it reflect C++ issues.

C is a relatively small language with a limited syntax set lacking many of the complex features of some other languages. Many of the complex features in C are not implemented as part of the language syntax, but rather implemented as library routines. As such, most of the available features in C are used relatively frequently.

Common use across a variety of languages may make some features less obscure. Because of the unstructured code that is frequently the result of using goto’s, the goto statement is frequently restricted, or even outright banned, in some C development environments. Even though the goto is encountered infrequently and the use of it considered obscure, because it is fairly obvious as to its purpose and since its use is common to many other languages, the functionality of it is easily understood by even the most junior of programmers.

The use of a combination of features adds yet another dimension. Particular combinations of features in C may be used rarely together or fraught with issues if not used correctly in combination. This can cause unexpected results and potential vulnerabilities.

### 6.54.2 Guidance to language users

* Consider the guidelines in TR 24772-1 clause 6.55.5.
* (Organizations) Specify coding standards that restrict or ban the use of features or combinations of features that have been observed to lead to vulnerabilities in the operational environment for which the software is intended.

## 6.55 Unspecified Behaviour [BQF]

### 6.55.1 Applicability of language

 This subclause requires a complete rewrite to have it reflect C++ issues.

The C standard has documented, in Annex J.1, 54 instances of unspecified behaviour. Examples of unspecified behaviour are:

* The order in which the operands of an assignment operator are evaluated
* The order in which any side effects occur among the initialization list expressions in an initializer
* The layout of storage for function parameters

Reliance on a particular behaviour that is unspecified leads to portability problems because the expected behaviour may be different for any given instance. Many cases of unspecified behaviour have to do with the order of evaluation of subexpressions and side effects. For example, in the function call

 f1(f2(x), f3(x));

 the functions f2 and f3 may be called in any order possibly yielding different results depending on the order in which the functions are called.

### 6.55.2 Guidance to language users

* Follow the guidelines of TR 24772-1 clause 6.56.5.
* Do not rely on unspecified behaviour because the behaviour can change at each instance. Thus, any code that makes assumptions about the behaviour of something that is unspecified should be replaced to make it less reliant on a particular installation and more portable.

## 6.56 Undefined Behaviour [EWF]

### 6.56.1 Applicability to language

This subclause requires a complete rewrite to have it reflect C++ issues.

The C standard does not impose any requirements on undefined behaviour. Typical undefined behaviours include doing nothing, producing unexpected results, and terminating the program.

The C standard has documented, in Annex J.2, 191 instances of undefined behaviour that exist in C. One example of undefined behaviour occurs when the value of the second operand of the / or % operator is zero. This is generally not detectable through static analysis of the code, but could easily be prevented by a check for a zero divisor before the operation is performed. Leaving this behaviour as undefined lessens the burden on the implementation of the division and modulo operators.

Other examples of undefined behaviour are:

* Referring to an object outside of its lifetime
* The conversion to or from an integer type that produces a value outside of the range that can be represented
* The use of two identifiers that differ only in non-significant characters

Relying on undefined behaviour makes a program unstable and non-portable. While some cases of undefined behaviour may be consistent across multiple implementations, it is still dangerous to rely on them. Relying on undefined behaviour can result in errors that are difficult to locate and only present themselves under special circumstances. For example, accessing memory deallocated by free() or realloc() results in undefined behaviour, but it may work most of the time.

### 6.56.2 Guidance to language users

* Follow the guidelines of TR 24772-1 clause 6.57.5.

## 6.57 Implementation–defined Behaviour [FAB]

### 6.57.1 Applicability to language

This subclause requires a complete rewrite to have it reflect C++ issues.

The C standard has documented, in Annex J.3, 112 instances of implementation-defined behaviour. Examples of implementation-defined behaviour are:

* The number of bits in a byte
* The direction of rounding when a floating-point number is converted to a narrower floating-point number
* The rules for composing valid file names

Relying on implementation-defined behaviour can make a program less portable across implementations. However, this is less true than for unspecified and undefined behaviour.

The following code shows an example of reliance upon implementation-defined behaviour:

unsigned int x = 50;

x += (x << 2) + 1; // x = 5x + 1

Since the bitwise representation of integers is implementation-defined, the computation on x will be incorrect for implementations where integers are not represented in two’s complement form.

### 6.57.2 Guidance to language users

* Follow the guidelines of TR 24772-1 clause 6.58.5.
* Eliminate to the extent possible any reliance on implementation-defined behaviour from programs in order to increase portability. Even programs that are specifically intended for a particular implementation may in the future be ported to another environment or sections reused for future implementations.

## 6.58 Deprecated Language Features [MEM]

### 6.58.1 Applicability to language

This subclause requires a complete rewrite to have it reflect C++ issues.

C deprecated one function, the function gets() and removed it from the standard in 2011.

C has deprecated several language features primarily by tightening the requirements for the feature:

* Implicit int declarations are no longer allowed.
* Functions cannot be implicitly declared. They must be defined before use or have a prototype.
* The use of the function ungetc() at the beginning of a binary file is deprecated.
* A return without expression is not permitted in a function that returns a value (and vice versa).

(NOTE) The deprecation of aliased array parameters has been removed, hence array parameters may be aliased.

### 6.58.2 Guidance to language users

* Follow the guidelines of TR 24772-1 clause 6.59.5.
* 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.

## 6.59 Concurrency – Activation [CGA]

### 6.59.1 Applicability to language

This subclause requires a complete rewrite to have it reflect C++ issues.

The C standard, in clause 7.26.5.1, requires a conforming implementation to set specific return codes to indicate whether or not a thread activation succeeded. Although the vulnerability does not apply to the C language, there could exist an application vulnerability if a program fails to check the return codes and take appropriate action.

### 6.59.2 Guidance to language users

* Follow the guidelines of TR 24772-1 clause 6.60.5.

## 6.60 Concurrency – Directed termination [CGT]

### 6.60.1 Applicability to language

This subclause requires a complete rewrite to have it reflect C++ issues.

## 6.60.1 Applicability to language

## 6.60.2 Guidance to language users

## 6.61 Concurrent Data Access [CGX]

### 6.61.1 Applicability to language

This subclause requires a complete rewrite to have it reflect C++ issues.

As stated in clause 5.1.2.4 of the C standard, a program that contains a data race exhibits undefined behaviour. In addition to threads, signal handlers also pose a risk of concurrent data access. It is the responsibility of the application to use atomic variables or mutexes to ensure that one thread or signal handler cannot modify an object while another thread or signal handler is attempting to access the same object.

### 6.61.2 Guidance to language users

* Follow the guidelines of TR 24772-1 clause 6.62.5.
* Use atomic variables where appropriate to avoid data races.
* Use mutexes appropriately to protect accesses to non-atomic shared objects.

## 6.62 Concurrency – Premature Termination [CGS]

### 6.62.1 Applicability to language

This subclause requires a complete rewrite to have it reflect C++ issues.

This vulnerability applies to C because the standard does not provide a mechanism to determine whether a thread has terminated.

### 6.62.2 Guidance to language users

* Follow the guidelines of TR 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.

## 6.63 Protocol Lock Errors [CGM]

### 6.63.1 Applicability to language

This subclause requires a complete rewrite to have it reflect C++ issues.

The C standard does not provide hidden protocols. Although the vulnerability does not apply to the C language, there could exist an application vulnerability if a program uses synchronization mechanisms incorrectly. For example:

atomic int a;

int b;

/\* . . . \*/

a += b; // This operation is an atomic read-modify-write of the variable ‘a’.

a = a + b; // This statement contains two accesses to ‘a’ and is not atomic.

### 6.63.2 Guidance to language users

* Follow the guidelines of TR 24772-1 clause 6.64.5.
* Be aware of the operation of each synchronization mechanism, such as the cases where accesses to atomic variables may occur more than once in a statement.

## 6.64 Uncontrolled Format String [SHL]

### 6.64.1 Applicability to language

This subclause requires a complete rewrite to have it reflect C++ issues.

### 6.64.2 Guidance to language users

[TBD]

# 7. Language specific vulnerabilities for C

[TBD]

# 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[[5]](#footnote-5).
* 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[[6]](#footnote-6).

[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. Integer types, Floating types and Pointer types are collectively called scalar types in the C Standard [↑](#footnote-ref-1)
2. An alias is a variable or formal parameter that refers to the same location as another variable or formal parameter. [↑](#footnote-ref-2)
3. Ada has developed a standard for interfacing with C. Fortran has included a Clause 15 that explains how to call C functions. [↑](#footnote-ref-3)
4. For example, Fortran and Ada specify how to call C functions. [↑](#footnote-ref-4)
5. This has been addressed by WG 14 in an optionally normative annex in the current working paper [↑](#footnote-ref-5)
6. The first edition should not be used or quoted in this work. [↑](#footnote-ref-6)