ISO/IEC JTC 1/SC 22/WG23 N0740

Date: 2017-08-17

ISO/IEC TR 24772–9

Edition 1

ISO/IEC JTC 1/SC 22/WG 23

Secretariat: ANSI

Deleted: 691

Deleted: 2

Deleted: 0

Deleted: 9

Formatted: zzCover, Space After: 100 pt, Tabs:Not at 0 cm

Deleted:

Formatted: Font:12 pt

Information Technology — Programming languages — Guidance to avoiding vulnerabilities in programming languages – Part 9 – Vulnerability descriptions for the programming language C++

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.

Document type: International standard
Document subtype: if applicable
Document stage: (10) development stage
Document language: E

Deleted: xxx

Contents

Page

Deleted: 4
Deleted: 4
Deleted: 6
Deleted: 6
Deleted: 6
Deleted: 7
Deleted: 8
Deleted: 10
Deleted: 12
Deleted: 12
Deleted: 14
Deleted: 14
Deleted: 15
Deleted: 17

Deleted: XXX

Deleted: 29
Deleted: 34
Deleted: 35
Deleted: 40
Deleted: 41
Deleted: 41
Deleted: 42
Deleted: 42
Deleted: 42
Deleted: 45
Deleted: 48

Deleted: xxx

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 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

Comment [CP1]:
Suggest there C++ terms need definitions

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] 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.

---

[1] Integer types, Floating types and Pointer types are collectively called scalar types in the C Standard

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.

Comment [CP2]:
All these C definitions need to be reviewed to decide which are still needed

4. Language concepts

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:

Deleted: . ... [1]

- 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.

Index		Reference
1	Make casts explicit in the return value of malloc. Example: *s = (struct foo*)malloc(sizeof(struct foo));* uses the C type system to enforce that the pointer to the allocated space will be of a type that is appropriate for the size. Because malloc returns a void *, without the cast, "s" could be of any random pointer type, with the cast, that mistake will be caught	[HFC]
2	Use bounds checking interfaces from Annex K of C11[4] in favour of non-bounds checking interfaces, such as strcpy_s instead of strcpy.	[HCB]
3	Use commonly available functions such as the POSIX functions htonl(), htons(), ntohl() and ntohs() to convert from host byte order to network byte order and vice versa	[STR]
4	Use stack guarding add-ons to detect overflows of stack buffers (REMOVE?)	[HCB]
5	Perform range checking before copying memory (using mechanisms such as memcpy and memmove), unless it can be shown that a range error cannot occur. Bounds checking is not performed automatically, but 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.	[XYW]
6	Check that a pointer is not null before dereferencing, unless it can be shown that the pointer is not null.	[XYH]
7	After a call to free as illustrated in the following code: free (ptr); ptr = NULL; Set the pointer to null to prevent multiple deallocation or use of a dangling reference via this pointer.	[XYK]

Comment [CP4]:
Needs to be reworked for C++, once section 6 is complete

8	Do not read uninitialized memory, including memory allocated by functions such as malloc.	[LAV]
9	Check that the result of an operation on an unsigned integer value will cause wrapping, unless it can be shown that wrapping cannot occur. Any of the following operators have the potential to wrap: a + b a – b a * b a++ a-- a += b a -= b a *= b a << b a<<=b -a	[FIF]
10	Check if the result of an operation on a signed integer value will cause an overflow, unless it can be shown that overflow cannot occur. Any of the following operators have the potential to overflow, which is undefined behavior in C: a + b a – b a * b a/b a%b a++ a-- a += b a -= b a *= b a /= b a %= b a << b a <<= b -a	
11	Ensure that a type conversion results in a value that can be represented in the resulting type.	[FLC]

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

Since C++ contains almost all of the C language as a subset, the type system, vulnerabilities and mitigations are as described in TR 24772-3, Clause 6.2.

In addition to the vulnerabilities and mitigations of C described in TR 24772-3, C++ adds specific casts which provide a number of (mostly) compile-time checks, so prevent casting between obviously inappropriate types.

- static casts
- const casts; and
- dynamic casts;

6.2.2 Guidance to language users

- Follow the advice provided in TR 24772-3 clause 6.2.2.
- 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*

6.3 Bit Representations [STR]

6.3.1 Applicability to language

C++ uses the bit representation mechanisms of C, as documented in TR 24772-3 clause 6.3.1.

6.3.2 Guidance to language users

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

-

Deleted:

Formatted: Normal, No bullets or numbering

Deleted: a number of feature relevant to a discussion of its type system: ... [2]

Formatted: Bulleted + Level: 1 + Aligned at: 0.63 cm + Indent at: 1.27 cm

Formatted: Not Highlight

Deleted: TR 24772-1 clause 6.2.5

Deleted: <#>Be aware of the rules for typing and conversions to avoid vulnerabilities. ... [3]

Formatted: Font:Italic

Deleted: u

Formatted: Font:Italic

Deleted: a

Formatted: Font:Italic

Deleted: l

Formatted: Font:Italic

Formatted: Font:Italic

Formatted: Font:Italic

Deleted: C++ supports a variety of sizes for integers such as short int, int, long int and long long int. Each may either be signed or unsigned. C++ also supports a variety of bitwise operators that make bit manipulations easy such as left and right shifts and bitwise operators. These bit manipulations can cause unexpected results or vulnerabilities through miscalculated shifts or platform dependent variations. ... [4]

Formatted: Normal, Space After: 0 pt

Formatted: Not Highlight

Formatted: Normal

Deleted: general

Deleted: TR 24772-1 clause 6.3.5

Deleted: :

Formatted: Font:(Default) Calibri

Formatted: List Paragraph, Bulleted + Level: 1 + Aligned at: 0.63 cm + Indent at: 1.27 cm, No widow/orphan control, Suppress line numbers, Don't allow hanging punctuation

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.

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

C++ offers *scoped enums* to replace the *enum* capability of C. Even if one uses C-stlye *enums*, C++ forbids implicit casts from integer to an enum, therefore preventing A = B+C, where A, B and C are variables of the same enum.

In C++, there is not a bidirectional cast between objects of an **enum class** and **int**, i.e. there is no implicit conversion from an integer type back to the enum type, hence operations such as "++", "+", "<" and enumerations used as array indexes are unavailable unless explicitly declared in the program. Hence, the general vulnerability of ... is avoided..

Idea that the enumerated type can have a user-specified underlying type for enumerated constants

6.5.2 Guidance to language users

- Use *scoped enumerations* in preference to the C-style *unscoped enumerations* for related values.
  - See the guidance of <CPP Core Guidelines Enum.4 and Enum.6 ...>
- 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 a function-style cast with braces instead of C-style or static casts

```
    e_type{7};
```

  - 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<typename std::underlying_type<e_type>::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.

Deleted: <#>Only use bitwise operators on unsigned integer values as the results of some bitwise operations on signed integers are implementation defined. ... [5]

Deleted: C++ permits the floating-point data types float, double and long double. Due to the approximate nature of floating-point representations, the use of float and double data types in situations where equality is needed or where rounding could accumulate over multiple iterations could lead to unexpected results and potential vulnerabilities in some situations. ... [6]

Deleted: In addition to the

Deleted: TR 24772-1 clause 6.4.5

Deleted: :

Deleted: <#>Do not use a floating-point expression in a Boolean test for equality. In C, implicit casts may make an expression floating-point even though the programmer did not expect ... [7]

Formatted: Font:Not Bold, Italic

Formatted: Font:Bold

Deleted: The enum type in C comprises a set of named integer constant values as in the example: ... [8]

Formatted: Font:+Theme Headings (Cambria)

Formatted: Font:+Theme Headings (Cambria), Not Highlight

Formatted: Font:Courier, 9 pt

Deleted: enum abc {A,B,C,D,E,F,G,H} var_abc; ... [9]

Formatted: Font:+Theme Headings (Cambria), Not Highlight

Formatted: Not Highlight

Formatted: Font:+Theme Headings (Cambria), Not Highlight

Formatted: Not Highlight

Formatted: Font:+Theme Headings (Cambria), Not Highlight

Formatted: Not Highlight

Formatted: Font:+Theme Headings (Cambria), Not Highlight

Formatted: Font:+Theme Headings (Cambria)

Formatted: Font:Not Italic

Formatted

Formatted: Not Highlight

Deleted: <#>

Formatted: Font:(Default) +Theme Body (Calibri), 11 pt, English (US), Highlight

Formatted: Font:Italic

Formatted: Indent: Left: 2.06 cm

Formatted: Font:+Theme Body (Calibri), Highlight

Formatted: Highlight

Formatted: Indent: Left: 2.62 cm, No bullets or numbering

Formatted: Font:(Default) Courier, 9 pt

Deleted: In addition to the general advice of TR 24772-1 clause 6.4.5: ... [10]

Formatted: Normal, Indent: Left: 0 cm

Formatted: Font:(Default) Courier New, Kern at 14 pt

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'

Formatted: Not Highlight

Deleted: C++ permits implicit conversions. That is, C++ will automatically perform a conversion without an explicit cast. For instance, ... [11]

Formatted: Not Highlight

Formatted: Font:Courier

Formatted: Not Highlight

Deleted: A loss of data (truncation) can occur when converting from a signed type to a signed type with less precision. For example, the following code can result in truncation: ... [12]

Formatted: Not Highlight

Deleted: Ch

Formatted: Not Highlight

Deleted: <#>eck the value of a larger type before converting it to a smaller type to see if the value in the larger type is within the range of the smaller type. Any conversion from a type with larger precision to a smaller precision type could potentially result in a loss of data. In some instances, this loss of precision is desired. Such cases should be explicitly acknowledged in comments. For example, the following code could be used to check whether a conversion from an unsigned integer to an unsigned character will result in a loss of precision; ... [13]

Deleted: <#>Close attention should be given to all warning messages issued by the compiler regarding multiple casts. Making a cast in C++ explicit will both remove the warning and acknowledge that the change in precision is on purpose. ... [14]

Deleted: that can take a single parameter

Deleted: to prevent accidental implicit conversion from the parameter type to the class type, unless such conversions are required

6.7 String Termination [CJM]

6.7.1 Applicability to language

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.

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.
In C++, the 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, so the following code:

```
    int foo(const int i) {
        int x[] = {0,0,0,0,0,0,0,0,0,0};
        return x[i];
    }
```

will return whatever is in location x[i] even if, i were equal to -10 or 10 (assuming either subscript was still within the address space of the program). This could be sensitive information or even a return address, which if altered by changing the value of x[-10]or x[10], could change the program flow.

The following code is more appropriate and would not violate the boundaries of the array x:

```
int foo( const int i) {
int x[X_SIZE] = {0};
if (i < 0 || i >= X_SIZE) {
  return ERROR_CODE;
  }
```

```
else {
  return x[i];
  }
}
```

A buffer boundary violation may also occur when copying, initializing, writing or reading a buffer if attention to the index or addresses used are not taken.

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

Vectors can be used as shown for arrays.

6.8.2 Guidance to language users

- For the use of C-style arrays, follow the guidance 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 iterators and range-based for-loops
- Use std::vector to access arrays of dynamic changing size
- When manually accessing array elements by indexing or pointer arithmetic, use bounds checking access such as array::at, unless it can be conclusively shown that the access can never be outside the bounds of the array.

  If bound checking each access would be prohibitively slow. If for performance reasons, index checking on each access is inappropriate, provide a check to show that no access will be outside the bounds of the array, e.g. when processing all the elements of a large array, show or check that the first and last elements to be accessed are in bounds.
- *Use boiler plate words about static analysis tools*
- *(Clive to polish)*

6.9 Unchecked Array Indexing [XYZ]

6.9.1 Applicability to language

C does not perform bounds checking on arrays, so though arrays may be accessed outside of their bounds, the value returned is undefined and in some cases may result in a program termination. For example, in C the following code is valid, though, for example, if i has the value 10, the result is undefined:

```
int foo(const int i) {
```

Deleted: vector
Deleted: I
Deleted: ay
Deleted: vector
Deleted: array(10
Deleted:)
Deleted: ay
Deleted: 1
Deleted: ay
Deleted: 1
Deleted: ay
Deleted: 1
Deleted: ay
Deleted: 1
Deleted: ~~For example, in the following move operation there is a buffer boundary violation:~~ ... [15]
Deleted: Use
Deleted: vector
Deleted: Always use bound checking access, such as vector::at. This guidance can only be ignored if it is clear that no access can ever be outside the bounds of the array (e.g. a fixed size array, with all indexing in-bounds),
Formatted: Indent: Left: 1.27 cm, No bullets or numbering
Deleted: or i
Deleted: ... [16]
Deleted: ... [17]
Formatted: Strikethrough
Formatted: Font:Not Italic
Formatted: Space After: 0 pt
Comment [CP12]: My inclination is to ditch all this, and refer back to 6.8 [HCB]

```
        int t;
        int x[] = {0,0,0,0,0};
        t = x[i];
        return t;
    }
```

The variable t will likely be assigned whatever is in the location pointed to by `x[10]` (assuming that `x[10]` is still within the address space of the program).

6.9.2 Guidance to language users

- Perform range checking before accessing an array since C does not perform bounds checking automatically. 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.
- Use the safer and more secure functions for string handling from the normative annex K of C11 [4], *Bounds-checking interfaces*. These are alternative string handling library functions. The functions verify that receiving buffers are large enough for the resulting strings being placed in them and ensure that resulting strings are null terminated.

6.10 Unchecked Array Copying [XYW]

6.10.1 Applicability to language

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

- Use classes, such as std::vector, that provide copy functions that ensure the target array is large enough for the indicated source, in preference to C library functions such as `memcpy()` or `memmove()`.
- Perform range checking before calling a memory copying function such as `memcpy()` and `memmove()`. These functions do not perform bounds checking automatically. 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.
- ~~Use the safer and more secure functions for string handling from the normative annex K of C11 [4], Bounds-checking interfaces.~~

6.11 Pointer Type Conversions [HFC]

6.11.1 Applicability to language

C++ allows casting the value of a pointer to and from another data type. These conversions can cause unexpected changes to pointer values.

Pointers in C++ refer to a specific type, such as integer. If `sizeof(int)` is 4 bytes, and `ptr` is a pointer to integers that contains the value 0x5000, then `ptr++` would make `ptr` equal to 0x5004. However, if `ptr` were a pointer to char, then `ptr++` would make `ptr` equal to `0x5001`. It is the difference due to data sizes coupled with conversions between pointer data types that cause unexpected results and potential vulnerabilities. Due to arithmetic operations, pointers may not maintain correct memory alignment or may operate upon the wrong memory addresses.

~~In particular, make casts explicit in the return value of malloc~~
~~Example: `s = (struct foo*)malloc(sizeof(struct foo));`~~
~~This uses the C type system to enforce that the pointer to the allocated space will be of a type that is appropriate for the size. Because `malloc` returns a `void *`, without the cast, `s` could be of any random pointer type; with the cast, that mistake will be caught~~

In general casting pointers breaks the type system and should be avoided. If it is unavoidable, use static_cast rather than reinterpret_cast. This is because reinterpret_cast simple treats the unmodified pattern of bits in the pointer as being of the target type rather than the original, but the C++ standard recognizes that the compiler may impose constrains or additional data requirements on a pointer. With static_cast, the compiler is allowed to make appropriate changes to the resulting pointer.

One common use of pointer conversion in C is to specify the actual type of the void* pointer returned by malloc when allocating memory on the heap, as in: (T*)malloc(sizeof(T));
Whilst malloc (and free) is still available in C++, memory allocation in C++ should be done using the new (and delete) keywords, as in: new T; // always returns a T* pointer

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

```
class Base { … };
class Derived: public Base { … };
```

Anywhere a Base* pointer is required, a pointer to a Derived class object can be used instead. In effect, there is an implicit cast of the Derived* pointer to Base*. This is called 'upcasting'. Sometimes, having got a Base* pointer, it may be necessary to convert it back to the derived type, 'downcasting'. This should be done using dynamic_cast, as this will check (at runtime) that the pointer is to an object of the correct type. If it's not, either NULL will be returned, or an error exception thrown:

```
class Base { … };
class Derived1: public Base { … };
class Derived2: public Base { … };

void foo(Base *ptr);  // forward reference
```

```
    Derived2 d2;
    foo(&v2);     // &v2 of type Derived2* implicitly upcast to Base*

void foo(Base *ptr)
    { Derived1 *p1 = dynamic_cast<{ Derived1*>(ptr);   // p1 becomes NULL, as ptr not a Devived1*
      Derived2 *p2 = dynamic_cast<{ Derived2*>(ptr);   // p2 become &v2
    }
```

6.11.2 Guidance to language users

- Follow the advice provided by TR 24772-1 clause 6.11.5.
- Cast between pointers using static_cast rather than reinterpret_cast, unless downcasting
- When downcasting, use dynamic_cast, and be aware that the result may be NULL
- ~~Maintain the same type to avoid errors introduced through conversions~~.
- ~~Always cast the value returned by `malloc` to an appropriate type~~
- 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

Comment [CP13]:
This seems pointless, as we are doing pointer conversion, so deliberately not maintaining the same type

6.12 Pointer Arithmetic [RVG]

6.12.1 Applicability to language

When performing pointer arithmetic in C, the size of the value to add to a pointer is automatically scaled to the size of the type of the pointed-to object. For instance, when adding a value to the byte address of a 4-byte integer, the value is scaled by a factor 4 and then added to the pointer. The effect of this scaling is that if a pointer P points to the i-th element of an array object, then (P) + N will point to the i+n-th element of the array. Failing to understand how pointer arithmetic works can lead to miscalculations that result in serious errors, such as buffer overflows.

In C, arrays have a strong relationship to pointers. The following example will illustrate arithmetic in C involving a pointer and how the operation is done relative to the size of the pointer's target. Consider the following code snippet:

```
int buf[5];
int *buf_ptr = buf;
```

where the address of `buf` is `0x1234`, after the assignment `buf_ptr` points to `buf[0]`. Adding 1 to `buf_ptr` will result in `buf_ptr == 0x1238` on a host where an `int` is 4 bytes; `buf_ptr` will then point to `buf[1]`. Not realizing that address operations will be in terms of the size of the object being pointed to can lead to address miscalculations and undefined behaviour.

6.12.2 Guidance to language users

- 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

C allows memory to be dynamically allocated primarily through the use of `malloc()`, `calloc()`, and `realloc()`. Each will return the address to the allocated memory. Due to a variety of situations, the memory allocation may not occur as expected and a null pointer will be returned. Other operations or faults in logic can result in a memory pointer being set to null. Using the null pointer as though it pointed to a valid memory location can cause a segmentation fault and other unanticipated situations.

Space for 10000 integers can be dynamically allocated in C in the following way:

```
int *ptr = malloc(10000*sizeof(int));  // allocate space for 10000 ints
```

`malloc()` will return the address of the memory allocation or a null pointer if insufficient memory is available for the allocation. It is good practice after the attempted allocation to check whether the memory has been allocated via an if test against `NULL`:

```
if (ptr != NULL)   // check to see that the memory could be allocated
```

Memory allocations usually succeed, so neglecting this test and using the memory will usually work. That is why neglecting the null test will frequently go unnoticed. An attacker can intentionally create a situation where the memory allocation will fail leading to a segmentation fault.

Faults in logic can cause a code path that will use a memory pointer that was not dynamically allocated or after memory has been deallocated and the pointer was set to null as good practice would indicate.

6.13.2 Guidance to language users

- Create a specific check that a pointer is not null before dereferencing it. As this can be expensive in some cases (such as in a `for` loop that performs operations on each element of a large segment of memory), judicious checking of the value of the pointer at key strategic points in the code is recommended.

6.14 Dangling Reference to Heap [XYK]

6.14.1 Applicability to language

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 to see that the memory could be allocated */
```

```
                /* perform some operations on the dynamic memory */
    free (ptr);    /* memory is no longer needed, so free it */
                /* program continues 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

- 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

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

- 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

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

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

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

- 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’, ‘I’ (lower case ‘L’), ‘l’ (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

Because C is an imperative language, programs in C can contain dead stores. This can result from an error in the initial design or implementation of a program, or from an incomplete or erroneous modification of an existing program.

A store into a volatile-qualified variable generally should not be considered a dead store because accessing such a variable may cause additional side effects, such as input/output (memory-mapped I/O) or observability by a debugger or another thread of execution.

6.18.2 Guidance to language users

- Use compilers and analysis tools to identify dead stores in the program.
- 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

Variables may be declared, but never used when writing code or the need for a variable may be eliminated in the code, but the declaration may remain. Most compilers will report this as a warning and the warning can be easily resolved by removing the unused variable.

6.19.2 Guidance to language users

- Resolve all compiler warnings for unused variables. This is trivial in C as one simply needs to remove the declaration of the variable. Having an unused variable in code indicates that either warnings were turned off during compilation or were ignored by the developer.

6.20 Identifier Name Reuse [YOW]

6.20.1 Applicability to language

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

- 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

Does not apply to C because C requires unique names and has a single global namespace. A diagnostic message is required for duplicate names in a single compilation.

6.22 Initialization of Variables [LAV]

6.22.1 Applicability to language

Local, automatic variables can assume unexpected values if they are used before they are initialized. The C Standard specifies, "If an object that has automatic storage duration is not initialized explicitly, its value is indeterminate". In the common case, on architectures that make use of a program stack, this value defaults to whichever values are currently stored in stack memory. While uninitialized memory often contains zeros, this is not guaranteed. Consequently, uninitialized memory can cause a program to behave in an unpredictable or unplanned manner and may provide an avenue for attack.

Assuming that an uninitialized variable is 0 can lead to unpredictable program behaviour when the variable is initialized to a value other than 0.

Many implementations will issue a diagnostic message indicating that a variable was not initialized.

6.22.2 Guidance to language users

- Heed compiler warning messages about uninitialized variables. These warnings should be resolved as recommended to achieve a clean compile at high warning levels.

Do not use memory allocated by functions such as `malloc()` before the memory is initialized as the memory contents are indeterminate.

6.23 Operator Precedence and Associativity [JCW]

6.23.1 Applicability to language

Operator precedence and associativity in C are clearly defined.

Mixed logical operators are allowed without parentheses.

6.23.2 Guidance to language users

- 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

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

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

C allows the usual sources of dead code (described in 6.26) that are common to most conventional programming languages.

C uses some operators that can be confused with other operators. For instance, the common mistake of using an assignment operator in a Boolean test as in:

```
int a;
/* … */
if (a = 1)
…
```

can cause portions of code to become dead code, because the else portion of the if statement cannot be reached.

6.26.2 Guidance to language users

- Apply the guidance provided in TR 24772-1 clause 6.26.5.
- Eliminate dead code to the extent possible from C programs.
- Use compilers and analysis tools to assist in identifying unreachable code.
- Use "//" comment syntax instead of "/*…*/" comment syntax to avoid the inadvertent commenting out sections of code.
- Delete deactivated code from programs due to the possibility of accidentally activating it.

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.

C contains a switch statement of the form:

```
char abc;
/* … */
switch (abc) {
   case 1:
       sval = "a";
      break;
   case 2:
      sval = "b";
      break;
   case 3:
      sval = "c";
      break;
   default:
      printf ("Invalid selection\n");
 }
```

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.

6.27.2 Guidance to language users

- Apply the guidance provided in TR 24772-1 clause 6.27.5
- Only a direct fall through should be allowed from one case to another. That is, every nonempty case statement should be terminated with a break statement as illustrated in the following example:

```
int i;
/* … */
switch (i) {
```

```
            case 1:
            case 2:
                i++;  /* fall through from case 1 to 2 is permitted */
                break;
             case 3:
                j++;
             case 4:  /* fall through from case 3 to 4 is not permitted */
                      /* as it is not a direct fall through due to the  */
                       /* j++ statement */
           }
```

- 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.

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 section of code:

```
int foo(int a, const int *b) {
    int i=0;
    /* … */
     a = 0;
    for (i=0; i<10; i++);
    {
       a = a + b[i];
    }
}
```

At first it may appear that a will be a sum of the numbers `b[0]` to `b[9]`. However, even though the code is layed 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. In this case, this mistake 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.

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. For example:

```
int a,b,i;
```

```
  /* … */
  if (i == 10){
       a = 5;     /* this is correct */
       b = 10;
      }
else
    a = 10;
    b = 5;
```

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.

6.29 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,i;
for (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.

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.

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);
}
```

Strings in C are also another common source of errors in C due to the need to allocate space for and account for the string sentinel value. A common mistake is to expect to store an n length string in an n length array instead of length n+1 to account for the sentinel `'\0'`. Interfacing with other languages that do not use sentinel values in strings can also lead to an off by one error.

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 for C 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.

Looping one more or one less is usually detectable by good testing. Due to the structure of the C language, this may be the main way to avoid this vulnerability. Unfortunately some cases may still slip through the development and test phase and manifest themselves during operational use.

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.

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 C 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 C 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.

Restrict the use of `goto`, `continue`, `break`, `return` and `longjmp` to encourage more structured programming.

- Encourage the use of a single exit point from a function. At times, this guidance can have the opposite effect, such as in the case of an if check of parameters at the start of a function that requires the remainder of the function to be encased in the if statement in order to reach the single exit point. If, for example, the use of multiple exit points can arguably make a piece of code clearer, then they should be used. However, the code should be able to withstand a critique that a restructuring of the code would have made the need for multiple exit points unnecessary.

6.32 Passing Parameters and Return Values [CSJ]

6.32.1 Applicability to language

C uses *call by value* parameter passing. The parameter is evaluated and its value is assigned to the formal parameter of the function that is being called. A formal parameter behaves like a local variable and can be modified in the function without affecting the actual argument. 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) {
   int t = *x;
   *x = *y;
   *y = t;
}
```

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

- 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]. Aliases can be avoided by following the respective guidelines of TR 24772-1 Clause 6.32.5.

[2] An alias is a variable or formal parameter that refers to the same location as another variable or formal parameter.

6.33 Dangling References to Stack Frames [DCM]

6.33.1 Applicability to language

C allows the address of a variable to be stored in a variable. Should this variable's address be, for example, the address of a local variable that was part of a stack frame, then using the address after the local variable has been deallocated can yield unexpected behaviour as the memory will have been made available for further allocation and may indeed have been allocated for some other use. Any use of perishable memory after it has been deallocated can lead to unexpected results.

6.33.2 Guidance to language users

- Do not assign the address of an object to any entity which persists after the object has ceased to exist. This is done in order to avoid the possibility of a dangling reference. Once the object ceases to exist, then so will the stored address of the object preventing accidental dangling references. In particular, never return the address of a local variable as the result of a function call.
- Long lived pointers that contain block-local addresses should be assigned the null pointer value before executing a return from the block.

6.34 Subprogram Signature Mismatch [OTR]

6.34.1 Applicability to language

Functions in C may be called with more or less than the number of parameters the receiving function expects. However, most C compilers will generate a warning or an error about this situation. If the number of arguments does not equal the number of parameters, the behaviour is undefined. This can lead to unexpected results when the count or types of the parameters differs from the calling to the receiving function. If too few arguments are sent to a function, then the function could still pop the expected number of arguments from the stack leading to unexpected results.

C allows a variable number of arguments in function calls. A good example of an implementation of this is 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. This can be a useful feature for situations such as `printf()`, but the use of this feature outside of special situations can be the basis for vulnerabilities.

Functions may or may not be defined with a function definition. The function definition may or may not contain a parameter type list. If a function that accepts a variable number of arguments is defined without a parameter type list that ends with the ellipsis notation, the behaviour is undefined.

If the calling and receiving functions differ in the type of parameters, C will, if possible, do an implicit conversion such as the call to `sqrt()` that expects a double:

```
double sqrt(double)
```

the call:

```
root2 = sqrt(2);
```
coerces the integer 2 into the double value 2.0.

6.34.2 Guidance to language users

- Follow the guidelines of TR 24772-1 clause 6.34.5.
- Use a function prototype to declare a function with its expected parameters to allow the compiler to check for a matching count and types of the parameters.

Do not use the variable argument feature except in rare instances. The variable argument feature such as is used in `printf()` is difficult to use in a type safe manner.

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

The C standard does not include exception handling, therefore only error status will be covered.

C provides the include file <errno.h> that defines the macros `EDOM`, `EILSEQ` and `ERANGE`, which expand to integer constant expressions with type `int`, distinct positive values and which are suitable for use in `#if` preprocessing directives. C also provides the integer `errno` that can be set to a nonzero value by any library function (if the use of `errno` is not documented in the description of the function in the C Standard, `errno` could be used whether or not there is an error). Though these values are defined, inconsistencies in responding to error conditions can lead to vulnerabilities.

6.36.2 Guidance to language users

- 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.

Set `errno` to zero before a library function call in situations where a program intends to check `errno` before a subsequent library function call.
Use `errno_t` to make it readily apparent that a function is returning an error code. Often a function that returns an `errno` error code is declared as returning a value of type int. Although syntactically correct, it is not apparent that the return code is an `errno` error code. The normative Annex K from ISO/IEC 9899:2011 [4] introduces the new type `errno_t` in `<errno.h>` that is defined to be type `int`.

- 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.

- For each routine, document all error conditions, matching error detection and reporting needs, and provide sufficient information for handling the error situation.
- Use static analysis tools to detect and report missing or ineffective error detection or handling.
- When execution within a particular context encounters an error, finalize the context by closing open files, releasing resources and restoring any invariants associated with the context.

6.37 Fault Tolerance and Failure Strategies [REU]

6.37.1 Applicability to language

Check that this writeup is consistent with the new title and writeup from Part 1. Wait until Erhard has reprocessed [REU] in Part 1.

Choosing when and where to exit is a design issue, but choosing how to perform the exit may result in the host being left in an unexpected state. C provides several ways of terminating a program including `exit()`, `_Exit()`, and `abort()`. A return from the initial call to the main function is equivalent to calling the `exit()` function with the value returned by the main function as its argument (this is if the return type of the main function is a type compatible with `int`, otherwise the termination status returned to the host environment is unspecified) or simply reaching the "}" that terminates the main function returns a value of 0.
All of the termination strategies in C have undefined, unspecified, and/or implementation defined behaviour associated with them. For example, if more than one call to the `exit()` function is executed by a program, the behaviour is undefined. The amount of clean-up that occurs upon termination such as the removal of temporary files or the flushing of buffers varies and may be implementation defined.

A call to `exit()` or `_Exit()` will terminate a program normally. Abnormal program termination will occur when abort() is used to exit a program (unless the signal `SIGABRT` is caught and the signal handler does not return). Unlike a call to `exit()`, when either `_Exit()` or `abort()` are used to terminate a program, it is implementation defined as to whether open streams with unwritten buffered data are flushed, open streams are closed, or temporary files are removed. This can leave a system in an unexpected state.
C provides the function `atexit()` that allows functions to be registered so that at normal program termination, the registered functions will be executed to perform desired functions. C requires the capability to register *at least* 32 functions. Implementations expecting more than 32 registered functions may yield unexpected results.

6.37.2 Guidance to language users

- Follow the guidance of TR 24772-1 clause 6.37.5.

Use a return from the `main()` program as it is the cleanest way to exit a C program.

- Use `exit()` to quickly exit from a deeply nested function.
- Use `abort()` in situations where an abrupt halt is needed. If `abort()` is necessary, the design should protect critical data from being exposed after an abrupt halt of the program.
- Become familiar with the undefined, unspecified and/or implementation aspects of each of the termination strategies.

6.38 Type-breaking Reinterpretation of Data [AMV]

6.38.1 Applicability to language

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.

6.38.2 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.

6.39 Deep vs. Shallow Copying [YAN]

6.39.1 Applicability to language

[TBD] *Stephen's thoughts. C does not have the classic OO deep copy problem, IMHO, but consider cases where A references a struct or array (which may contain references to deeper levels). B = A would simply copy the pointer (correct?) so the same issue can be there.*

[DMK] Not really. An array cannot be assigned to another array. Given an array object A and an array object B of the same type, B = A is a syntax error. Given array A and pointer P that points to objects of the type of A's elements, P = A copies a pointer to A, but the programmer already knows that because P was declared as a pointer. The problem in this section does not apply to arrays by themselves.

Given a struct object A and a struct object B of the same type, B = A copies the contents, not a pointer, so one level of deep copying is already done and is not a problem. If A contains a member that is a pointer, or a member that is an array, struct, or union that contains pointers, then there is a deep copy problem.

6.39.2 Guidance to language users

[TBD]

6.40 Memory Leak [XYL]

6.40.1 Applicability to language

C can allow memory leaks as many programs use dynamically allocated memory. C relies on manual memory management rather than a built in garbage collector primarily since automated memory management can be

unpredictable, impact performance and is limited in its ability to detect unused memory such as memory that is still referenced by a pointer, but is never used.

- Memory is dynamically allocated in C using the library calls `malloc()`, `calloc()`, and `realloc()`. When the program no longer needs the dynamically allocated memory, it can be released using the library call `free()`. Should there be a flaw in the logic of the program, memory continues to be allocated but is not freed when it is no longer needed. A common situation is where memory is allocated while in a function, the memory is not freed before the exit from the function and the lifetime of the pointer to the memory has ended upon exit from the function.

6.40.2 Guidance to language users

- Use debugging tools such as leak detectors to help identify unreachable memory.
- Allocate and free memory in the same module and at the same level of abstraction to make it easier to determine when and if an allocated block of memory has been freed.
- Use realloc() only to resize dynamically allocated arrays.
- Use garbage collectors that are available to replace the usual C library calls for dynamic memory allocation which allocate memory to allow memory to be recycled when it is no longer reachable. The use of garbage collectors may not be acceptable for some applications as the delay introduced when the allocator reclaims memory may be noticeable or even objectionable leading to performance degradation.

6.41 Templates and Generics [SYM]

This vulnerability does not apply to C, because C does not implement these mechanisms.

6.42 Inheritance [RIP]

This vulnerability does not apply to C, because C does not implement this mechanism.

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

This vulnerability does not apply to C, because C does not implement polymorphism.

6.44 Redispatching [PPH]

This vulnerability does not apply to C, because C does not implement this mechanism.

6.45 Polymorphic variables [BKK]

This vulnerability does not apply to C, because C does not implement this mechanism.

6.46 Extra Intrinsics [LRM]

This vulnerability does not apply to C, because C does not implement these mechanisms.

6.47 Argument Passing to Library Functions [TRJ]

6.47.1 Applicability to language

Parameter passing in C is either pass by reference or pass by value. There isn't a guarantee that the values being passed will be verified by either the calling or receiving functions. So values outside of the assumed range may be received by a function resulting in a potential vulnerability.

A parameter may be received by a function that was assumed to be within a particular range and then an operation or series of operations is performed using the value of the parameter resulting in unanticipated results and even a potential vulnerability.

6.47.2 Guidance to language users

- Follow the guidelines of TR 24772-1 clause 6.47.5.
- Do not make assumptions about the values of parameters.
- Do not assume that the calling or receiving function will be range checking a parameter. Therefore, establish a strategy for each interface to check parameters in either the calling or receiving routines.

6.48 Inter-language Calling [DJS]

6.48.1 Applicability to language

The C Standard defines the calling conventions, data layout, error handing and return conventions needed to use C from another language. Ada has developed a standard for interfacing with C. Fortran has included a Clause 15 that explains how to call C functions. Calls from C into other languages become the responsibility of the programmer.

6.48.2 Guidance to language users

- Follow the guidelines of TR 24772-1 clause 6.48.5.
- Minimize the use of those issues known to be error-prone when interfacing from C, such as
  1. passing character strings,
  2. dimension, bounds and layout issues of arrays,
  3. interfacing with other parameter formats such as call by reference or name,
  4. receiving return codes, and
  5. bit representation.

6.49 Dynamically-linked Code and Self-modifying Code [NYY]

6.49.1 Applicability to language

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.

6.49.2 Guidance to language users

- 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.

6.50 Library Signature [NSQ]

6.50.1 Applicability to language

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.

6.50.2 Guidance to language users

- 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.

6.51 Unanticipated Exceptions from Library Routines [HJW]

Since C does not have exceptions nor does it handle exceptions passed from other language systems, this vulnerability does not apply. See 6.36 for a discussion of Ignored errors. See TR 24772-1 clause 6.47 in the case where libraries written in languages that use exceptions may be called.

6.52 Pre-processor Directives [NMP]

6.52.1 Applicability to language

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.

6.52.2 Guidance to language users

This vulnerability can be avoided or mitigated in C in the following ways:

- 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.

6.53 Suppression of Language-defined Run-time Checking [MXB]

Does not apply to C since there are no language-defined runtime checks.

6.54 Provision of Inherently Unsafe Operations [SKL]

6.54.1 Applicability to language

C was designed for implementing system software where some unsafe operations are inherent and common.

6.54.2 Guidance to language users

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

6.55 Obscure Language Features [BRS]

6.55.1 Applicability of language

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.

6.55.2 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.

6.56 Unspecified Behaviour [BQF]

6.56.1 Applicability of language

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.

6.56.2 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.

6.57 Undefined Behaviour [EWF]

6.57.1 Applicability to language

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.

6.57.2 Guidance to language users

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

6.58 Implementation–defined Behaviour [FAB]

6.58.1 Applicability to language

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.

6.58.2 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.

6.59 Deprecated Language Features [MEM]

6.59.1 Applicability to language

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.

6.59.2 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.

6.60 Concurrency – Activation [CGA]

6.60.1 Applicability to language

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.

6.60.2 Guidance to language users

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

6.61 Concurrency – Directed termination [CGT]

6.61.1 Applicability to language

Does not apply to C because C does not implement this mechanism.

6.62 Concurrent Data Access [CGX]

6.62.1 Applicability to language

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.

6.62.2 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.

6.63 Concurrency – Premature Termination [CGS]

6.63.1 Applicability to language

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

6.63.2 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.

6.64 Protocol Lock Errors [CGM]

6.64.1 Applicability to language

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.
```

6.64.2 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.

6.65 Uncontrolled Format String [SHL]

6.65.1 Applicability to language

[TBD]

6.65.2 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[3].
- 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)

[3] This has been addressed by WG 14 in an optionally normative annex in the current working paper

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)[4].

[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

[4] The first edition should not be used or quoted in this work.

[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. 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: 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 (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

Page 5: [1] Deleted **Stephen Michell** **4/6/17 4:19:00 PM**

block-structured language: A language that has a syntax for enclosing structures between bracketed keywords, such as an if statement bracketed by if and endif, as in Fortran, or a code section bracketed by BEGIN and END, as in PL/1.

comb-structured language: A language that has an ordered set of keywords to define separate sections within a block, analogous to the multiple teeth or prongs in a comb separating sections of the comb. For example, in Ada, a block is a 4-pronged comb with keywords declare, begin, exception, end, and the if statement in Ada is a 4-pronged comb with keywords if, then, else, end if.[CP1]

Page 8: [2] Deleted **Stephen Michell** **4/6/17 1:57:00 PM**

a number of feature relevant to a discussion of its type system:

C++ reuses the keyword **static**, as a property of class member variables and functions. A static member variable is a variable for which there is only one copy accessible from all instances of objects of that class (c.f. non-static variables, where each class object has its own variable). A static member function is one that is guaranteed not to modify non-static class members (checked at compile time)

C++ also extends the concept of **const** to class member functions. A **const** class member function is guaranteed not to modify any non-static class member variables, unless they have the **mutable** qualifier (checked at compile time)

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. These

Page 8: [3] Deleted **Stephen Michell** **4/6/17 2:01:00 PM**

Be aware of the rules for typing and conversions to avoid vulnerabilities.

Do not cast to an inappropriate type.

Page 8: [4] Deleted **Stephen Michell** **4/6/17 2:12:00 PM**

C++ supports a variety of sizes for integers such as short int, int, long int and long long int. Each may either be [CP2]signed or unsigned. C++ also supports a variety of bitwise operators that make bit manipulations easy such as left and right shifts and bitwise operators. These bit manipulations can cause unexpected results or vulnerabilities through miscalculated shifts or platform dependent variations.

Bit manipulations are necessary for some applications and may be one of the reasons that a particular application was written in C++. Although many bit manipulations can be rather simple in C++, such as masking off the bottom three bits in an integer, more complex manipulations can cause unexpected results. For instance, right shifting a signed integer is implementation defined in C++, while shifting by

an amount greater than or equal to the size of the data type is undefined behaviour. For instance, on a host where an int is of size 32 bits,

```
unsigned int foo(const int k) {
    unsigned int i = 1;
    return i << k;
}
```

is undefined for values of k greater than or equal to 32.

The storage representation for interfacing with external constructs can cause unexpected results. Byte orders may be in little-endian or big-endian format and unknowingly switching between the two can unexpectedly alter values.

Page 9: [5] Deleted **Stephen Michell** **4/6/17 2:21:00 PM**

Only use bitwise operators on unsigned integer values as the results of some bitwise operations on signed integers are implementation defined.

Use the POSIX standard functions htonl(), htons(), ntohl() and ntohs() (where available) to convert from [CP3]host byte order to network byte order and vice versa. This would be needed to interface between an i80x86 architecture where the Least Significant Byte is first with the network byte order, as used on the Internet, where the Most Significant Byte is first. If these functions are not available, identify and use appropriate equivalent functions. Use bitwise operations only as a last resort.

In cases where there is a possibility that the shift is greater than the size of the variable, perform a check as the following example shows, or a modulo reduction before the shift:

```
unsigned int i;
unsigned int k;
unsigned int shifted_i;
…
    if (k < sizeof(unsigned int)*CHAR_BIT)
        shifted_i = i << k;
    else
        // handle error condition
```

Page 9: [6] Deleted **Stephen Michell** **4/6/17 2:22:00 PM**

C++ permits the floating-point data types float, double and long double. Due to the approximate nature of [CP4]floating-point representations, the use of float and double data types in situations where equality is needed or where rounding could accumulate over multiple iterations could lead to unexpected results and potential vulnerabilities in some situations.

As with most data types, C++ is flexible in how float, double and long double can be used. For instance, C++ allows the use of floating-point types to be used as loop counters and in equality statements. Even though a loop may be expected to only iterate a fixed number of times, depending on the values

contained in the floating-point type and on the loop counter and termination condition, the loop could execute forever. For instance iterating a time sequence using 10 nanoseconds as the increment:

```
float x;
for (x=0.0; x!=1.0; x+=0.00000001)
```

may or may not terminate after 10,000,000 iterations. The representations used for `x` and the accumulated effect of many iterations may cause `x` to not be identical to 1.0 causing the loop to continue to iterate forever.

Similarly, the Boolean test

```
float x=1.336f;
float y=2.672f;
if (x == (y/2))
```

may or may not evaluate to true. Given that `x` and `y` are constant values, it is expected that consistent results will be achieved on the same platform. However, it is questionable whether the logic performs as expected when a float that is twice that of another is tested for equality when divided by 2 as above. This can depend on the values selected due to the quirks of floating-point arithmetic.

Page 9: [7] Deleted **Stephen Michell** **4/6/17 2:24:00 PM**

- Do not use a floating-point expression in a Boolean test for equality. In C, implicit casts may make an expression floating-point even though the programmer did not expect it.
- Check for an acceptable closeness in value instead of a test for equality when using floats and doubles to avoid rounding and truncation problems.
- Do not convert a floating-point number to an integer unless the conversion is a specified algorithmic requirement or is required for a hardware interface.

Page 9: [8] Deleted **Stephen Michell** **4/7/17 11:11:00 AM**

The enum type in C comprises a set of named integer constant values as in the example:[CP5]

Page 9: [9] Deleted **Stephen Michell** **4/7/17 10:33:00 AM**

enum abc {A,B,C,D,E,F,G,H} var_abc;

The values of the contents of abc would be A=0, B=1, C=2, and so on. C allows values to be assigned to the enumerated type as follows:

enum abc {A,B,C=6,D,E,F=7,G,H} var_abc;

This would result in:

A=0, B=1, C=6, D=7, E=8, F=7, G=8, H=9

yielding both gaps in the sequence of values and repeated values.

If a poorly constructed enum type is used in loops, problems can arise. Consider the enumerated type abc defined above used in a loop:

```
 int x[8];
for (i=A; i<=H; i++){
   t = x[i];
}
```

Because the enumerated type abc has been renumbered and because some numbers have been skipped, the array will go out of bounds and there is potential for unintentional gaps in the use of x.

Page 9: [10] Deleted **Stephen Michell** **4/7/17 10:37:00 AM**

In addition to the general advice of TR 24772-1 clause 6.4.5:

- Use enumerated types in the default form starting at 0 and incrementing by 1 for each member if possible. The use of an enumerated type is not a problem if it is well understood what values are assigned to the members.
- Avoid using loops that iterate over an enum that has representation specified for the enums, unless it can be guaranteed that there are no gaps or repetition of representation values within the enum definition.
- Use an enumerated type to select from a limited set of choices to make possible the use of tools to detect omissions of possible values such as in switch statements.
- Use the following format if the need is to start from a value other than 0 and have the rest of the values be sequential:

```
enum abc {A=5,B,C,D,E,F,G,H} var_abc;
```

- Use the following format if gaps are needed or repeated values are desired and so as to be explicit as to the values in the enum, then:

```
enum abc {
    A=0,
    B=1,
    C=6,
    D=7,
    E=8,
    F=7,
    G=8,
    H=9
    } var_abc;
```

A common use of enum in C programs is to define a collection of unrelated integer constants, as this is regarded as more robust than a sequence of pre-processor #define statements. This should be avoided in C++, in favour of constant declarations that may be made at file or class scope, e.g.

```
static const unsigned int bufferLen  = 128;
```

Page 10: [11] Deleted **Stephen Michell** **4/7/17 11:46:00 AM**

C++ permits implicit conversions. That is, C++ will automatically perform a conversion without an explicit cast. For instance, [CP6]

```
int i;
float f=1.25f;
i = f;
```

This implicit conversion will discard the fractional part of f and set i to 1. If the value of f is greater than INT_MAX, then the assignment of f to i would be undefined.

The rules for implicit conversions are defined in the C++ standard. For instance, integer types smaller than int are promoted when an operation is performed on them. If all values of Boolean, character or integer type can be represented as an int, the value of the smaller type is converted to an int; otherwise, it is converted to an unsigned int.

Integer promotions are applied as part of the usual arithmetic conversions to certain argument expressions; operands of the unary +, -, and ~ operators, and operands of the shift operators. The following code fragment shows the application of integer promotions:

```
char c1, c2;
c1 = c1 + c2;
```

Integer promotions require the promotion of each variable (c1 and c2) to int size. The two int values are added and the sum is truncated to fit into the char type.

Integer promotions are performed to avoid arithmetic errors resulting from the overflow of intermediate values. For example:

```
signed char cresult, c1, c2, c3;
c1 = 100;
c2 = 3;
c3 = 4;
cresult = c1 * c2 / c3;
```

In this example, the value of c1 is multiplied by c2. The product of these values is then divided by the value of c3 (according to operator precedence rules). Assuming that signed char is represented as an 8-bit value, the product of c1 and c2 (300) cannot be represented. Because of integer promotions, however, c1, c2, and c3 are each converted to int, and the overall expression is successfully evaluated. The resulting value is truncated and stored in cresult. Because the final result (75) is in the range of the signed char type, the conversion from int back to signed char does not result in lost data. It is possible that the conversion could result in a loss of data should the data be larger than the storage location.

Page 10: [12] Deleted **Stephen Michell** **4/7/17 11:46:00 AM**

A loss of data (truncation) can occur when converting from a signed type to a signed type with less precision. For example, the following code can result in truncation:

```
signed long int sl = LONG_MAX;
```

```
signed char sc = (signed char)sl;
```

The C++ standard defines rules for integer promotions, integer conversion rank, and the usual arithmetic conversions. The intent of the rules is to ensure that the conversions result in the same numerical values, and that these values minimize surprises in the rest of the computation.

~~A recent innovation from ISO/IEC TR 24731-1 [13] that has been added to the C standard 9899:2011 [4] is the~~ [CP7]~~definition of the `rsize_t` type. Extremely large object sizes are frequently a sign that an object's size was calculated incorrectly. For example, negative numbers appear as very large positive numbers when converted to an unsigned type like `size_t`. Also, some implementations do not support objects as large as the maximum value that can be represented by type `size_t`. For these reasons, it is sometimes beneficial to restrict the range of object sizes to detect programming errors. For implementations targeting machines with large address spaces, it is recommended that `RSIZE_MAX` be defined as the smaller of the size of the largest object supported or `(SIZE_MAX >> 1)`, even if this limit is smaller than the size of some legitimate, but very large, objects. Implementations targeting machines with small address spaces may wish to define `RSIZE_MAX` as `SIZE_MAX`, which means that there is no object size that is considered a runtime-constraint violation.~~

Page 10: [13] Deleted **Stephen Michell** **4/7/17 12:14:00 PM**

eck the value of a larger type before converting it to a smaller type to see if the value in the larger type is within the range of the smaller type. Any conversion from a type with larger precision to a smaller precision type could potentially result in a loss of data. In some instances, this loss of precision is desired. Such cases should be explicitly acknowledged in comments. For example, the following code could be used to check whether a conversion from an unsigned integer to an unsigned character will result in a loss of precision:

```
unsigned int i;
unsigned char c;
…
if (i <= UCHAR_MAX) {  // check against the maximum value
                       // for an object of type unsigned char
  c = (unsigned char) i;
}
else {
  // handle error condition
}
```

Page 10: [14] Deleted **Stephen Michell** **4/7/17 12:14:00 PM**

Close attention should be given to all warning messages issued by the compiler regarding multiple casts. Making a cast in C++ explicit will both remove the warning and acknowledge that the change in precision is on purpose.

If mixed types are used in an expression, ensure that each conversion preserves the value before being used as an operand in another operation in the same expression

Page 12: [15] Deleted **Stephen Michell** **4/6/17 2:56:00 PM**

~~For example, in the following move operation there is a buffer boundary violation:~~[CP8]

```
char buffer_src[]={"abcdefg"};
char buffer_dest[5]={0};
strcpy(buffer_dest, buffer_src);
```

the buffer_src is longer than the buffer_dest, and the code does not check for this before the actual copy operation is invoked. A safer way to accomplish this copy would be:

```
char buffer_src[]={"abcdefg"};
char buffer_dest[5]={0};
strncpy(buffer_dest, buffer_src, sizeof(buffer_dest) -1);
```

this would not cause a buffer bounds violation, however, because the destination buffer is smaller than the source buffer, the destination buffer will now hold "abcd", the 5th element of the array would hold the null character.

Page 12: [16] Deleted	Stephen Michell	4/6/17 3:28:00 PM

I

Page 12: [17] Deleted	Stephen Michell	4/6/17 2:58:00 PM

- Use length restrictive functions such as strncpy() instead of strcpy().
- Use stack guarding add-ons to detect overflows of stack buffers.
- Do not use the deprecated functions or other language features such as gets().
- Be aware that the use of all of these measures may still not be able to stop all buffer overflows from happening. However, the use of them can make it much rarer for a buffer overflow to occur and much harder to exploit it.

Use the safer and more secure functions for string handling from the normative annex K of C11 [4], Bounds-checking interfaces. The functions verify that output buffers are large enough for the intended result and return a failure indicator if they are not. Optionally, failing functions call a *runtime-constraint handler* to report the error. Data is never written past the end of an array. All string results are null terminated. In addition, these functions are re-entrant: they never return pointers to static objects owned by the function. Annex K also contains functions that address insecurities with the C input-output facilities.