ISO/IEC JTC 1/SC 22/OWGV N 0043

Adam Schofield & Clive Pygott, "A Tabulation of the unpredictable features of the C++ language," September 2006, QINETIQ/S&DU/TIM/CR060019

Date	2006-09-19
Contributed by	Clive Pygott
Original file name	C++ Vulnerabilities Report - v03.pdf
Notes	

QinetiQ

A Tabulation of the unpredicatable features of the C++ language

QINETIQ/S&DU/TIM/CR060019

Cover + vi + 28 pages
September 2006
Adam Schofield & Clive Pygott

Customer Information	
Customer Reference Number	MCIP 03 06 001 12
Task Title	Safety Critical Systems: Robust Languages
Customer Contact Name	D. Wharf
Staff Requirement/Target	
Project Number	C/CIP/N03508
Milestone Number	12
Date Due (dd/mm/yyyy)	10/02/2006

Authorisation

Principal authors

Name	Adam Schofield & Clive Pygott
Appointment	Systems Assurance Group
Location	Woodward A109, QinetiQ, Malvern

Name	Prof C M O'Halloran
Appointment	Director Systems Assurance Group
	Woodward A109, QinetiQ, Malvern

Record of changes

Issue	Date	Detail of Changes
1	3/2/2006	First Issue
2	8/9/2006	Added Unique ID column to tables

Executive summary

In recent years there has been a growing interest in the use of C++ in high integrity applications. However, there is as yet no publicly recognised standard for the development of such code.

The primary requirement for any software that is to be used in a high-integrity environment is predictability. Typically, this requirement for predictability has led to the development of coding standards that aim to avoid or control 'problematic' areas of a language, such as the SPARK Ada and MISRA C subsets.

The starting point for the SPARK and MISRA C subsets were the annexes in the respective ISO standards that described the various unspecified, compiler dependent etc. language features. These provided a benchmark against which any proposed coding standard could be judged.

The ISO standard for C++ does not provide an equivalent annex of language vulnerabilities, so the aim of this report is to address this omission.

It should be noted that this report is solely concerned with core language features, and does not address any issues associated with libraries or support environments.

It is anticipated that this report may be used in two ways:

- as guidance to anyone developing a C++ reduced-risk subset, as to the language specification issues that need to be addressed,
- as a bench mark against which any proposed reduced-risk subset can be accessed (again for language specification issues)

In either event, the development of a reduced-risk language subset to address language specification issues is only part of the requirement for high integrity software development. The need to consider avoidance of common programmer errors, clarity of intent to aid maintenance and the development of tool support to police any subset and analyse developed code are outside the scope of this report.

It is recommended that this report is given wide circulation in an attempt to achieve public scrutiny and industrial consensus that the language specification issues that need to be addressed for the safety critical/related use of C++ have been identified.

It is also recommended that any proposed reduced-risk language subset should be assessed against the issues identified here.

List of contents

This page is intentionally blank

1 Introduction

1.1 Background

1.1.1 The primary requirement for any software that is to be used in a high-integrity environment is predictability. Typically, this requirement for predictability has led to the development of coding standards that aim to avoid or control 'problematic' areas of a language. For C and Ada, this has meant the definition of the SPARK [1] and MISRA C [2] subsets, which aim to:

- avoid unspecified[1] behaviour in the language
- avoid compiler dependent behaviour
- avoid 'confusing' behaviour associated with common programmer mistakes and increase the clarity of a program, to minimise mistakes during maintenance
- avoid constructs and features that cannot be adequately tested/analysed by the available technology

1.1.2 It can be argued that the first three should be ranked in that order, i.e. 'unspecified behaviour' being the most serious concern, etc. The argument being that:

- for unspecified behaviour it is impossible to predict what the program will do
- for compiler dependencies, it may be possible to predict what the program will do, but it may be difficult to demonstrate that the same behaviour will happen under all circumstances, and the implementation is not robust if the compiler changes
- for confusing behaviour etc., the behaviour of the program is well defined and may be what the programmer wants. Hence, any restrictions are precautionary.

1.1.3 For the fourth bullet, the impact depends upon what 'constructs and features' are being considered, and clearly the requirements may change with time, as testing/analysis technology develops.

1.2 Purpose of this Report

1.2.1 In recent years there has been a growing interest in the use of C++ in high integrity applications. However, there is as yet no publicly recognised standard for the development of such code.

1.2.2 The starting point for the SPARK subset was the annex in the ISO language definition standard [9] that described the various unspecified, compiler dependent etc. features of the language. For MISRA C it was the C standard's [8] annex and a number of books [6,7] that identified the equivalent features of that language. These provided a benchmark against which any proposed coding standard could be judged, at least as far as unspecified and compiler dependent behaviours were concerned (issues of clarity and testability require separate consideration).

1.2.3 The ISO standard for C++ [3] does not provide an equivalent annex of language vulnerabilities, so the aim of this report is to address this omission.

[1] using 'unspecified' its general sense, rather than the narrow definition provided in [3], c.f. 1.4.2

1.2.4 This report is solely concerned with core language features, and does not address any issues associated with libraries or support environments.

1.3 History and contractual

1.3.1 During 2004, staff at QinetiQ carried out a review of the C++ ISO standard [3], looking for keywords, such as 'unspecified', 'undefined', 'compiler dependent' etc. and assembled a list of vulnerabilities as reference [4].

1.3.2 Simultaneously and independently, a similar study was being conducted at the University of York [5].

1.3.3 This report, provides a consolidated list of the vulnerabilities identified in these two report, traceable back to the ISO standard.

1.3.4 It should be noted that these two source documents contain additional information relating to the use of C++ in critical systems, not covered in this report.

1.3.5 This report has been produced as part of an MoD Corporate Research Programme (CRP) on 'Robust Languages'. It represents the deliverable for task 3.1 'identify C++ vulnerabilities'. It aims to facilitate the development of C++ coding standards, such as being undertaken by MISRA as 'MISRA C++'.

1.4 Structure of the report

1.4.1 Section 2 contains further background information, a classification scheme for vulnerabilities (derived from the ISO standard [3]) and a description of the format of the tables in the rest of the report.

1.4.2 Sections 3 to 7 contain lists of the vulnerabilities for each of the five classifications:

- Unspecified[2]
- Undefined
- Implementation defined
- Indeterminate
- 'Behaviour that requires no diagnostic'

1.4.3 Section 8 is the conclusions and recommendations.

1.5 Linking this report to the C++ ISO standard

1.5.1 For each of the vulnerabilities described in sections 3 to 7, there is a hypertext link to the relevant page of the ISO standard (as well as a printed reference in terms of sub-section and paragraph number). When correctly configured, clicking on the link will open the standard on the correct page.

1.5.2 To make the hyperlinks work under Windows, you will need the following:

- This report

[2] from now on, 'unspecified' is used as defined in [3], which is distinct from 'undefined' and 'indeterminate' c.f. 1.1.1

- The ISO C++ Standard in PDF format, which can be purchased from the British Standards Organisation on-line store:

  http://www.bsonline.bsi-global.com/server/index.jsp
- The auto-extracting zip file *LinkedReport.exe*, available from the supplier of this report.

1.5.3 Note, due to copyright restrictions we are unable to distribute the ISO standard with this report (hence the need to purchase it separately).

1.5.4 Place *LinkedReport.exe* in the some folder and double click on *LinkedReport.exe*. This will extract the contents to a sub-folder named ‘CPP Vulnerabilities’. Copy this report into ‘CPP Vulnerabilities’.

1.5.5 ‘CPP Vulnerabilities’ contains a sub-folder called ‘data’, the contents of a which will be a series of HTM files that are accessed by this report’s links and cause the ISO standard to be opened in a PDF reader on the correct page. Move/copy the ISO C++ Standard PDF to this ‘data’ folder.

1.5.6 The ISO C++ Standard pdf should be named ‘ISO14882 - 2003.pdf’. The document should already have this name by default. If it is named differently, change the name by right-clicking on the document, selecting rename, and typing in "ISO14882 - 2003" (without quotes). Drag and drop the folder into the ‘data’ folder.

2 Comparison with C, and Classification of Issues

2.1 Comparison with the use of C in High Integrity Applications

2.1.1 In the early 1990's, C was regarded as a 'non-starter' for safety critical applications. For example, the wide and varied use of pointers and the ability to manipulate them was seen as providing far too many ways of generating unexpected aliasing and dependencies, and so lead to unexpected results.

2.1.2 A major step forward for the use of C in safety applications was the production of the book "Safer C" by Les Hatton [6] in 1994. In this Hatton identified all the undefined, unspecified, implementation defined etc. features that could be found in the C language standard and then analysed how they could be detected and what means of verification could be used to show that programs avoid these known issues.

2.1.3 The outcome from Hatton's book has been that the holes and pitfalls of C were opened up for scrutiny. Researchers were then able to look towards ways of developing programs which could avoid the known language deficiencies. A consequence from this initial work was that in 1998 the UK Motor Industry Software Reliability Association produced a set of guideline rules [1] for aiding the development of safety related automotive applications. This document soon became the de-facto standard in many organisations around the world and formed a solid starting point for take up by the safety critical software community.

2.1.4 The MISRA guidelines have proved to be a significant advancement over the way C programs are viewed for safety related applications. The lessons learnt from the initial uses of the MISRA guidelines have been incorporated into the recently released revised version.

2.1.5 However, just defining a reduced-risk sub-set for a language is not, in itself, sufficient to guarantee that all dangerous language issues have been avoided. This is especially true for C with its use and manipulation of pointers. Unlike Ada it is not possible to ban the use of all pointers as they are central to efficient C based programs. Therefore in order to minimise the risks it is necessary to capture the use of C within a robust fault management framework. The same is expected to be true of C++, as it is a direct descendant of C.

2.2 Classification of C++ Language Standard Issues

2.2.1 Whilst the ISO Ada and C language standards [8,9] provide a clear list of those language features which are unspecified, undefined, etc., the C++ language standard [3] does not do this. This report is therefore the result of reviewing the C++ language standard, to draw out such features. Sections 3 to 7 of this report provide detailed tables of the different categories of features that need to be considered. In overview these are:

- Unspecified behaviour
- Undefined behaviour
- Implementation defined behaviour
- Indeterminate behaviour
- Behaviour that requires no diagnostic

2.2.2 **Unspecified behaviour** is defined in the C++ language standard as: "*behaviour, for a well-formed program construct and correct data, that depends on the implementation. The implementation is not required to document which behaviour occurs. It should be noted that usually, the range of possible behaviours is delineated by the language Standard*".

2.2.3 **Undefined behaviour** is defined in the C++ language standard as: "*behaviour, such as might arise upon use of an erroneous program construct or erroneous data, for which the language standard imposes no requirements. Undefined behaviour may also be expected when the language standard omits the description of any explicit definition of behaviour. It should be noted that permissible 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).*"

2.2.4 **Implementation defined behaviour** is defined in the C++ language standard as "*behaviour, for a well-formed program construct and correct data, that depends on the implementation and that each implementation shall document*".

2.2.5 **Indeterminate behaviour** is defined in the C++ language standard through negative statements. For example, some language statements define that a construct shall not use a particular feature. It is therefore left indeterminate what would happen if such a construct did use a particular feature.

2.2.6 "**Behaviour that requires no diagnostics**" are features of the language which do not follow the required or expected rules but for which the language standard states that no diagnostic information is required to be given to the user. Thus it is possible that these issues could be violating the language definition yet no information is passed to a programmer that such a violation has occurred.

2.2.7 Sections 3 to 7 provide details of the different occurrences of the above categories that have been determined from a review of the C++ language standard. The tables provided in the sections give cross references to the appropriate sections of the language standard where a fuller description of the language features and issue can be found.

2.2.8 Overall the number of issues found within the C++ language standard for which reduced-risk coding rules will be required are shown in table 2.1. Note that these are core language issues, and exclude issues relating to libraries etc.

Category	Language issues
Unspecified behaviour	31
Undefined behaviour	81
Implementation behaviour	62
Indeterminate behaviour	5
Behaviour that requires no diagnostic	19

Table 2.1: *C++ language issues*

2.2.9 As a rough comparison, the equivalent analysis for C identified a total of 12 unspecified behaviours, 54 undefined and 41 implementation dependent behaviours.

2.3 Classification by language feature

2.3.1 Each of the following sections includes a table with a format similar to that shown in table 2.2, where each row represents a specific issue (unspecified feature etc).

Unique ID	ISO Standard Reference	ISO Standard Paragraph	Description	Classification	
1.01	3.6.2	2	Whether an object is fully, or merely zero-initialized when an object refers to another object of namespace scope with static storage duration potentially requiring dynamic initialization and defined later in the same translational unit.	Initialisation Order.	Link

Table 2.2: *C++ language sample issue*

2.3.2 The first two columns provide a reference into the C++ language reference [3], in terms of a sub-section number and paragraph within the sub-section. The third column is a short description of the issue.

2.3.3 If the report has been installed as described in section 1.5, the “Link” in the final column allows the C++ ISO standard to be ‘opened’ on the appropriate page.

2.3.4 The fourth column classifies the issue into broad ‘areas of concern’. The areas of concern used are shown in table 2.3

Classification	**Description**
Casting	Issues involving explicit type conversion with cast operators
Constant Objects	Issues involving objects that can not be modified, i.e. objects with a const-qualification
Enumerated Types	Issues involving the value and type of enumeration constants
Evaluation	Issues related to evaluation, but not its order, e.g. whether or how many times expressions are evaluated, rather than in what order, c.f. Initialisation
Evaluation Order	Issues relating to order of evaluation of sub-expression within an expression etc. That is, the elements being ordered are visible in the program, c.f. Initialisation Order
Exceptions	Issues relating to any ‘exceptional’ behaviour. This does not just relate to the explicit C++ exception mechanism
Execution Environment	Issues involving freestanding environments ("execution takes place without the benefit of an operating system") and the **main** function
Function Calls	Issues relating to calling functions
Inheritance	Issues relating to inheritance (excluding virtual functions), both

	single and multiple
Initialisation	Issues relating to initialisation, excluding the order of initialisation, c.f. Evaluation
Initialisation Order	Order of execution of initialisation actions. That is, where the elements being ordered or the action of concern is implied (e.g. program start) rather than explicit, c.f. Evaluation Order
Layout	Layout of objects in memory, e.g. the order and relative position of sub-objects within an object, c.f. Representation
Lexical Analysis	Issues relating to lexical analysis of the source text
Memory Allocation	Issues relating to if and how memory is allocated and deallocated
Mixed Language Working	Issues relating to the use of multiple language linkages
NameSpace	Issues relating to name-spaces in the general computer science sense of the scope of a name, rather than necessarily to do with C++'s namespace construct
Object Lifetime	Issues relating to the start and end of an object's lifetime and constructor/destructor calls, e.g. when (or if) an object is created or destroyed
One Definition Rule	Issues relating to the One Definition Rule over multiple translation units, as defined in [3, section 3.2]
Pointers	Issues relating to pointer types
Pre-processor	Issues relating to macros and pre-processing tokens
Representation	The representation of an object in memory (e.g. 2's compliment vs. sign and magnitude), c.f. Layout
String Literal	Issues relating to string literals
Template	Issues relating to templates
Type Info	Issues relating to types, type_info objects and typeid expressions
Value Range	Issues relating to the range of values a type can take
Virtual Functions	Issues relating to virtual functions and calls

Table 2.3: *Classification definitions used in the following sections*

3 Unspecified Behaviours

Unspecified behaviour is defined in the C++ language standard as "*behaviour, for a well-formed program construct and correct data, that depends on the implementation. The implementation is not required to document which behaviour occurs*." [1.3.13]

Unique ID	ISO Standard Reference	ISO Standard Paragraph	Description	Classification	
1.01	3.6.2	2	Whether an object is fully, or merely zero-initialized when an object refers to another object of namespace scope with static storage duration potentially requiring dynamic initialization and defined later in the same translational unit.	Initialisation Order.	Link
1.02	3.7.3.1	2	The order, contiguity and initial value of storage allocated by the allocation functions.	Representation / Layout.	Link
1.03	5	4	The order of evaluation of operands of individual operators and subexpressions of individual expressions, and the order in which side effects take place.	Evaluation Order	Link
1.04	5.2.2	8	The order of evaluation of arguments in a function call and the order of evaluation of the postfix expression and the argument expression list.	Evaluation Order / Function Calls	Link
1.05	5.2.8	1	Whether or not the destructor is called for the *type_info* object at the end of the program.	Object Lifetime / Type Info	Link
1.06	5.2.9	7	An integer type is explicitly converted to an enumeration type but the integral value is not within the range of the enumeration values	Casting / Enumerated Types.	Link
1.07	5.2.10	6	A pointer to a function is explicitly converted to a function of a different type using reinterpret_cast.	Casting / Pointers / Function Calls	Link
1.08	5.2.10	7	A pointer to an object is explicitly converted to a pointer to an object of a different type using *reinterpret_cast*.	Casting / Pointers	Link
1.09	5.2.10	9	A pointer to member of some type is explicitly converted to a pointer to another member of another type using *reinterpret_cast*.	Casting / Pointers	Link
1.10	5.3.4	21	The order of evaluation of the allocation function and its arguments.	Evaluation Order/ Initialisation Order	Link
1.11	5.3.4	21	The evaluation of arguments if the allocation function returns null or exits using an exception.	Evaluation Order.	Link

1.12	5.4	6	Whether the *static_cast* or *reinterpret_cast* interpretation is used if either the operand or destination type of the cast is a pointer to incomplete class type.	Casting	Link
1.13	5.9	2	Pointers are compared using a relational operator that do not point to members of the same object, elements of the same array or to the same functions, etc…	Memory Allocation / Layout / Pointers	Link
1.14	5.10	2	Pointers are compared using an equality operator and either is a pointer to a virtual member function.	Memory Allocation / Layout / Pointers	Link
1.15	7.2	4	The type of an uninitialised first enumerator.	Enumerated Types	Link
1.16	7.2	4	The value of an uninitialised enumerator is not representable in the type of the preceding enumerator.	Enumerated Types	Link
1.17	7.2	9	A value is not in the range of the enumeration type to which it is explicitly converted.	Casting / Enumerated Types.	Link
1.18	8.3.2	3	Whether a reference requires storage.	Memory Allocation / Layout	Link
1.19	8.3.6	9	The order of evaluation of function arguments.	Evaluation Order / Function Calls	Link
1.20	9.2	12	The order of allocation of nonstatic data members separated by an *access-specifier*	Memory Allocation / Layout	Link
1.21	10	3	The order in which the base class subobjects are allocated in the most derived object	Memory Allocation / Layout / Inheritance	Link
1.22	11.1	2	The order of allocation of data members with separate access-specifier labels	Memory Allocation / Layout	Link
1.23	12.1	15	The value of an object obtained, if during the construction of a *const* object, the object is accessed through an lvalue not obtained from the constructor's *this* pointer.	Initialisation Order	Link
1.24	12.2	5	The order of creation of temporary objects.	Evaluation Order.	Link
1.25	12.8	13	Whether subobjects representing virtual base classes are assigned more than once by the implicitly-defined copy assignment operator.	Evaluation / Inheritance	Link
1.26	14.7.1	5	Whether the instantiation occurs when the overload resolution process can determine the correct function to call without instantiating a class template definition.	Evaluation / Template	Link
1.27	14.7.1	9	Whether an implementation implicitly instantiates a virtual member function of a class template if the virtual member function would not otherwise be instantiated.	Memory Allocation / Layout / Template	Link

1.28	15.1	4	The way memory is allocated for the temporary copy of an exception being thrown	Memory Allocation / Layout / Exceptions	Link
1.29	15.1	4	Deallocation of memory for a temporary object when the last handler exits by any means other than a throw and the temporary object is then destroyed.	Memory Allocation / Layout / Exceptions	Link
1.30	16.3.2	2	The order of evaluation of # and ## operators.	Evaluation Order / Pre-processor	Link
1.31	16.3.3	3	The order of evaluation of ## operators.	Evaluation Order / Pre-processor	Link

4 Undefined Behaviours

Undefined behaviour is defined in the C++ language standard as "*behaviour, such as might arise upon use of an erroneous program construct or erroneous data, for which this International Standard imposes no requirements. Undefined behaviour may also be expected when this International Standard omits the description of any explicit definition of behaviour. [Note: permissible 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). Many erroneous program constructs do not engender undefined behaviour; they are required to be diagnosed .]*" [1.3.12]

Unique ID	ISO Standard Reference	ISO Standard Paragraph	Description	Classification	
2.01	2.1	2	A character sequence that matches a universal-character-name is produced due to the splicing of physical source lines in the translation process.	Lexical Analysis	Link
2.02	2.1	2	A non empty source file does not end in a new line character, or ends in a new line character immediately preceded by a backslash character.	Lexical Analysis	Link
2.03	2.1	4	A character sequence that matches a universal-character-name is produced due to token concatenation.	Lexical Analysis	Link
2.04	2.4	2	An unmatched ' or a " character is encountered on a logical source line during tokenisation.	Lexical Analysis	Link
2.05	2.8	2	The characters ', \, ", /*, or // are encountered between the < and > delimiters or the characters ', \, /*, or // are encountered between the " delimiters in the two forms of a header name preprocessing token.	Lexical Analysis / Pre-processor	Link
2.06	2.13.1	2	An integer literal cannot be represented by any of the allowed types.	Value Range	Link
2.07	2.13.2	3	The character following a backslash does not give a valid escape sequence.	Lexical Analysis	Link
2.08	2.13.4	2	An attempt is made to modify a string literal.	String Literal / Constant Objects	Link
2.09	2.13.4	3	A narrow string literal token is adjacent to a wide string literal token.	String Literal	Link
2.10	3.2	5	The behaviour of a program if two definitions in separate translation units do not satisfy the one definition rule.	One Definition Rule	Link

2.11	3.6.1	4	The library function exit is called to end a program during the destruction of an object with static storage duration.	Object Lifetime	Link
2.12	3.6.3	2	A function contains a local object of static storage duration that has been destroyed and the function is called during the destruction of an object with static storage duration and the flow of control passes through the definition of the previously destroyed object.	Object Lifetime	Link
2.13	3.7.3.1	2	The results of dereferencing a pointer returned as a request for zero size space in a call to an allocation function.	Memory Allocation / Pointers	Link
2.14	3.7.3.2	4	Attempt to use a pointer to a deleted object.	Memory Allocation / Pointers	Link
2.15	3.8	4	The side effects of a non-trivial destructor of an object of class type whose lifetime has ended, but whose destructor has not been called explicitly.	Object Lifetime	Link
2.16	3.8	5	An object will be or was of a class type with a non-trivial destructor and the pointer is used as the operand of a *delete-expression*.	Object Lifetime / Pointers	Link
2.17	3.8	5	Series of uses of a pointer to a non-POD class type between object storage allocation and the start of object lifetime, and the end of object lifetime and storage deallocation.	Object Lifetime / Pointers	Link
2.18	3.8	6	An lvalue-to-rvalue conversion is applied to an lvalue that refers to an object whose lifetime has not yet started but whose storage has been allocated, or whose lifetime has ended but whose storage has not been reused or released.	Memory Allocation / Object Lifetime	Link
2.19	3.8	6	Series of uses of an lvalue that refers to a non-POD class type between object storage allocation and the start of object lifetime, and the end of object lifetime and storage deallocation.	Memory Allocation / Object Lifetime	Link
2.20	3.8	8	A program ends the lifetime of an object of type T with static or automatic storage duration, T has a non-trivial destructor and an object of a different type occupies the storage location when the implicit destructor call takes place.	Object Lifetime	Link
2.21	3.8	9	A new object is created at the storage location that a *const* object with static or automatic storage duration occupies or, at the storage location that such a *const* object used to occupy before its lifetime ended.	Memory Allocation / Object Lifetime / Constant Object	Link

2.22	3.10	15	A program attempting to access the stored value of an object through an lvalue of other than one of the types specified.	Casting	Link
2.23	4.1	1	An lvalue, which does not refer to an object of type T or is uninitialised, is used where an rvalue of type T is expected.	Casting	Link
2.24	4.8	1	A floating-point conversion produces a result that cannot be represented in the space provided.	Casting / Value Range	Link
2.25	4.9	1	A floating-integral conversion produces a result that cannot be represented in the space provided	Casting / Value Range	Link
2.26	5	4	An object is modified more than once or is modified and accessed other than to determine the new value, between two sequence points.	Evaluation Order	Link
2.27	5	5	An arithmetic operation is invalid (such as division or modulus by zero) or produces a result that cannot be represented in the space provided (such as overflow or underflow).	Value Range	Link
2.28	5.2.2	1	A function is called through an expression whose function type has a language linkage that is different from the language linkage of the function type of the called function's definition.	Mixed Language Working / Function Calls	Link
2.29	5.2.2	7	An argument with no parameter, after standard conversions, has a non-POD class type.	Function Calls / Variable Length Parameter List	Link
2.30	5.2.9	5	A *static_cast* is used to cast an lvalue of class type to a non-derived class.	Casting	Link
2.31	5.2.9	8	A *static_cast* is used to cast a pointer of class type to a pointer from a non-derived class.	Casting / Pointers	Link
2.32	5.2.9	9	A *static_cast* is used to cast a pointer to a class member to a pointer to a member of a non-derived class	Casting / Pointers	Link
2.33	5.2.10	6	A pointer to a function is converted by *reinterpret_cast* to point to a function of a different type and used to call a function of a type not compatible with the original type.	Casting / Function Calls	Link
2.34	5.2.11	7	Depending on the type of object, a write operation through the pointer, lvalue or pointer to data member resulting from a const_char that casts away a const-qualifier may produce undefined behaviour.	Layout / Casting / Constant Object	Link
2.35	5.2.11	12	The use of values produced from conversions between pointers and functions, pointers and member functions and in particular a pointer to a *const* member function to a pointer to a non-const member function.	Casting / Function Calls / Constant Object	Link

2.36	5.3.1	4	The address of an object with incomplete type, whose complete type declares *operator&()* as a member function.	Overloading / Pointers	Link
2.37	5.3.4	6	The first array dimension applied to a *new* operator is negative.	Memory Allocation	Link
2.38	5.3.5	2	The behaviour of the *delete* operator on a pointer to a non-array object or a pointer to a sub-object representing the base class of such an object that was not obtained from a *new* operator.	Object Lifetime	Link
2.39	5.3.5	2	The value of the operand of delete is not the pointer value that resulted from a previous array new-expression when deleting an array.	Object Lifetime	Link
2.40	5.3.5	3	When deleting an object and the static type of the operand is different from its dynamic type and either the static type is not a base class of the operand's dynamic type, or the static type does not have a virtual destructor.	Object Lifetime	Link
2.41	5.3.5	3	The dynamic type of the object to be deleted differs from its static type when deleting an array.	Object Lifetime	Link
2.42	5.3.5	5	The object being deleted has incomplete class type at the point of deletion and the complete class has a non-trivial destructor or deallocation function.	Object Lifetime	Link
2.43	5.5	4	In a pointer-to-member operation the dynamic type of an object does not contain the member to which the pointer refers.	Layout / Pointers	Link
2.44	5.5	6	The second operand of an ->* expression is the null pointer to a member value.	Pointers	Link
2.45	5.6	4	The second operand of the / or % operators is zero.	Value Range	Link
2.46	5.7	5	A pointer that does not behave like a pointer to an element of an array object is added to or subtracted from.	Layout / Value Range / Pointer	Link
2.47	5.7	5	The resultant pointer from an addition or subtraction to a pointer to an element of an array which does not point within the array (or one beyond).	Layout / Value Range / Pointer	Link
2.48	5.7	6	Two pointers to elements of the same array object are subtracted, the result does not fit in the space provided and there is an arithmetic overflow.	Value Range / Pointer	Link
2.49	5.7	6	Pointers that do not behave like pointers to elements of the same array are subtracted.	Layout / Pointer	Link
2.50	5.8	1	An expression is shifted by a negative number or by an amount greater than or equal to the width in bits of the expression being shifted.	Value Range	Link
2.51	5.17	8	An object is assigned to an overlapping object.	Layout	Link

2.52	6.6.3	2	The effect of flowing off the end of a function that is expected to return a value	Function Calls	Link
2.53	6.7	4	Control re-enters a declaration recursively while an object is being initialized.	Initialisation	Link
2.54	7.1.5.1	4	An attempt is made to modify a *const* object, other than any class member declared mutable.	Constant Objects	Link
2.55	7.1.5.1	7	An attempt is made to refer an object defined with volatile-qualified type through the use of an lvalue with non-volatile-qualified type.	Casting	Link
2.56	8.3.2	4	Dereferencing a null pointer.	Pointers	Link
2.57	9.3.1	1	A member function of a class X is called for an object that is not of type X or a type derived from X.	Casting / Function Calls	Link
2.58	10.4	6	A virtual call is made from a constructor (or destructor) of an abstract class to a pure virtual function directly or indirectly for the object being created (or destroyed).	Object Lifetime / Virtual Functions	Link
2.59	12.4	12	A destructor is invoked for an object that is not of the destructor's class or not of a class derived from the destructor's class.	Object Lifetime / Casting	Link
2.60	12.4	14	A destructor is invoked for an object whose lifetime has ended	Object Lifetime	Link
2.61	12.6.2	8	A member function (including virtual member functions) is called for an object under construction, or an object under construction is used as the operand of the typeid operator or of a *dynamic_cast* performed in a ctor-initializer (or a function called directly or indirectly from a ctor-initializer) before all of the mem-initializers for base classes have been completed.	Evaluation Order / Object Lifetime / Inheritance	Link
2.62	12.7	1	Referring to any nonstatic member or base class of an object of non-POD class type, before the constructor begins execution and after the destructor finishes execution.	Evaluation Order / Object Lifetime	Link
2.63	12.7	2	Converting a pointer to an object of class X to a direct or indirect base class of X, where the construction of the object has not started or the destruction of the object has completed.	Object Lifetime / Pointer	Link
2.64	12.7	2	Forming a pointer to (or access the value of) a direct nonstatic member of an object, where the construction of the object has not started or the destruction of the object has completed.	Object Lifetime	Link
2.65	12.7	3	The result of making a virtual call using an explicit class member access and the object expression refers to the object under construction or destruction but its type is neither the constructor or destructor's own class or one of its bases.	Virtual Functions / Object Lifetime / Inheritance	Link

2.66	12.7	4	The operand of *typeid* refers to an object under construction or destruction and the static type of the operand is neither the constructor or destructor's class nor one of its bases.	Object Lifetime / Type Info	Link
2.67	12.7	5	If the operand of the *dynamic_cast* refers to the object under construction or destruction and the static type of the operand is not a pointer to or object of the constructor is not a pointer to or object of the constructor or destructor's own class or one of its bases.	Casting / Object Lifetime	Link
2.68	14.6.4.2	1	If a function call that depends on a template parameter would be ill-formed or would find a better match had the lookup within the associated namespaces considered all the function declarations with external linkage introduced with those namespaces in all translation units	Template / NameSpace / Function Calls	Link
2.69	14.7.1	14	The instantiation of a template produces recursion beyond some defined limit	Template	Link
2.70	15.3	10	Referring to any nonstatic member or base class of an object in the handler for a *function-try-block* of a constructor or destructor for that object.	Exceptions / Object Lifetime	Link
2.71	15.3	16	Flowing off the end of a *function-try-block* in a value returning function.	Exceptions	Link
2.72	16.1	4	The token defined is generated during the expansion of a #if or #elif pre-processing directive.	Pre-processor	Link
2.73	16.1	4	The *#defined* pre-processing directive does not match one of the two specified forms	Pre-processor	Link
2.74	16.2	4	The *#include* pre-processing directive that results after expansion does not match one of the header name forms.	Pre-processor	Link
2.75	16.3	10	A function-like macro argument consists of no pre-processing tokens.	Pre-processor	Link
2.76	16.3	10	There are sequences of pre-processing tokens within the list of function-like macro arguments that would otherwise act as pre-processing directive lines.	Pre-processor	Link
2.77	16.3.2	2	The result of the pre-processing operator # is not a valid character string literal.	Pre-processor / String Literal	Link
2.78	16.3.3	3	The result of the pre-processing concatenation operator ## is not a valid pre-processing token.	Pre-processor	Link
2.79	16.4	3	The #line pre-processing directive specifies zero or a number greater than 32767.	Pre-processor	Link
2.80	16.4	5	The #line pre-processing directive that results after expansion does not match one of the two well-defined forms.	Pre-processor	Link

2.81	16.8	3	One of the following identifiers is the subject of a *#define* or a *#undef* pre-processing directive. *__LINE__*, *__FILE__*, *__DATE__*, *__TIME__*, *__STDC__*, *__cplusplus*, or the identifier *defined*.	Pre-processor	Link

5 Implementation Defined Behaviours

Implementation-Defined behaviour is defined in the C++ language standard as "*behaviour, for a well formed program construct and correct data, that depends on the implementation and that each implementation shall document*." [1.3.5]

Unique ID	ISO Standard Reference	ISO Standard Paragraph	Description	Classification	
3.01	2.1	1	The mapping of physical source file characters.	Lexical Analysis	Link
3.02	2.1	3	Whether each non-empty sequence of white-space characters other than new line is retained or replaced by one space character.	Lexical Analysis	Link
3.03	2.1	8	Whether the source of the translational units containing the definitions of the templates for the requisite instantiations is required to be available.	Lexical Analysis / Template	Link
3.04	2.2	3	The values of the members of the execution character sets.	Value Range / Representation	Link
3.05	2.8	1	The mapping of the sequences in both forms of *header-names*. See 16.2(2)	Pre-processor	Link
3.06	2.13.2	1	The value of a multi-character literal.	Value Range / Representation	Link
3.07	2.13.2	2	The value of a wide-character literal containing multiple *c-chars*.	Value Range / Representation	Link
3.08	2.13.2	4	The value of a character literal that falls outside of the implementation defined range for *char* or *w_char*.	Value Range / Representation	Link
3.09	2.13.2	5	The encoding of a universal-character-name where the execution character set has no encoding for the character named.	Value Range / Representation	Link
3.10	2.13.3	1	The actual value used for a floating literal whose value is not in the range of representable values for its type.	Value Range / Representation	Link
3.11	2.13.4	2	Whether all string literals are distinct (stored in non-overlapping objects).	Layout / String Literals	Link
3.12	3.6.1	1	Whether a program in a freestanding environment is required to define a *main* function.	Execution Environment	Link
3.13	3.6.1	1	Start-up and termination in a freestanding environment.	Execution Environment	Link

3.14	3.6.1	2	The type of the *main* function, though its return type must be *int*.	Execution Environment	Link
3.15	3.6.1	3	The linkage of *main*.	Execution Environment	Link
3.16	3.6.2	3	Whether the dynamic initialization of an object of namespace scope is done before the first statement of main.	Initialisation	Link
3.17	3.9	4	For POD types, the set of values of which the value representation (a set of bits in the object representation that determines a *value*) is one discrete element.	Memory Allocation / Layout	Link
3.18	3.9	5	The packing needed between sub-objects to meet alignment requirements	Memory Allocation / Layout	Link
3.19	3.9.1	1	Whether *char* is equivalent to *unsigned char* or *signed char*.	Value Range	Link
3.20	3.9.1	2	Size of *int*.	Value Range	Link
3.21	3.9.1	5	Type of *wchar_t*.	Value Range / Representation	Link
3.22	3.9.1	8	The value representation of floating-point types.	Value Range / Representation	Link
3.23	3.9.2	3	The value representation of pointer types.	Value Range / Representation	Link
3.24	4.7	3	The value of a signed integer type due to the conversion from either an integer or an enumeration type when the value cannot be represented in the destination type.	Casting / Value Range	Link
3.25	4.8	1	The value resulting from converting a value of a floating point type to another floating point type that cannot exactly represent the original value	Casting / Representation	Link
3.26	4.9	2	The choice of either the next higher or lower representable value when an rvalue of an integer or enumeration type is converted to an rvalue of a floating-point type but exact conversion is not possible.	Casting	Link
3.27	5.2.8	1	The class (*name*) derived from *std::type_info* of an lvalue of dynamic type constname, that is the result of a *typeid* expression.	Type Info	Link
3.28	5.2.10	3	The mapping performed by *reinterpret_cast*.	Casting	Link
3.29	5.2.10	4	The mapping function used to explicitly converting a pointer to any integral type large enough to hold it.	Casting	Link

3.30	5.2.10	5	Mappings between pointers and integers other than when a value of integral or enumeration type is explicitly converted into a pointer or when a pointer is converted to an integer of sufficient size and back to the same pointer type.	Casting	Link
3.31	5.3.3	1	The result of *sizeof* applied to any fundamental type (other than *char*, *signed char* and *unsigned char)*, in particular *sizeof(bool)* and *sizeof(wchar_t).*	Representation	Link
3.32	5.6	4	The sign of the remainder using the binary % operator unless both operands are non-negative.	Value Range / Representation	Link
3.33	5.7	6	The signed integral type given as a result of the subtraction of two pointers to elements of the same array object.	Value Range	Link
3.34	5.8	3	The value given as a result of >> shift operator where the *shift-expression* has a signed type and is negative	Value Range	Link
3.35	7.1.5.2	1	Whether bit-fields and objects of char type are represented as signed or unsigned quantities.	Value Range / Representation	Link
3.36	7.2	5	The integral type used as the *underlying type* for an enumeration.	Value Range / Representation	Link
3.37	7.4	1	The meaning of an asm declaration.	Mixed Language Working	Link
3.38	7.5	1	Implementation specific properties associated with an entity with language linkage	Mixed Language Working	Link
3.39	7.5	2	The meaning of the *string-literal* in a *linkage-specification*	Mixed Language Working / String Literal	Link
3.40	7.5	2	The spelling of the language's name when the *string-literal* in a *linkage-specification* names a programming language	Mixed Language Working	Link
3.41	7.5	2	The semantics of a language linkage other than C++ or C.	Mixed Language Working	Link
3.42	7.5	9	Linkage from C++ to objects defined in other languages and to objects defined in C++ from other languages.	Mixed Language Working	Link
3.43	8.5.3	8	How the reference is bound when a reference to type “cv1 T1” is initialized by an expression “cv2 T2”.	Initialisation	Link
3.44	9.6	1	The allocation of bit-fields within a class.	Layout / Representation	Link
3.45	9.6	1	Alignment of bit-fields	Layout	Link

3.46	9.6	3	Whether a plain (neither explicitly signed nor unsigned) *char*, *short*, *int* or *long* bit-field is signed or unsigned.	Value Range / Representation	Link
3.47	14	4	The linkage of a template, a template explicit specialization or a class template partial specialization, if it is something other than C or C++.	Mixed Language Working / Template	Link
3.48	14.7.1	14	The limit on the total depth of recursive instantiation of templates	Template	Link
3.49	15.3	9	Whether or not the stack is unwound before the call to *terminate(),* in the case where no matching handler is found in a program.	Exceptions	Link
3.50	15.5.2	2	The object of type *std::bad_exception* that is used to replace an exception thrown or rethrown by the *unexpected()* function that the *exception-specification* does not allow.	Exceptions	Link
3.51	16.1	4	Whether the value of an interpreted character literal matches the value obtained when an identical character literal occurs in an expression.	Pre-processor	Link
3.52	16.1	4	Whether a single-character character literal may have a negative value.	Pre-processor	Link
3.53	16.2	2	The sequence of places searched for the header file specified between the < and > delimiters due to a *#include <h-char-sequence> new-line* pre-processing directive.	Pre-processor	Link
3.54	16.2	2	During execution of a *#include* pre-processor directive, how the places are searched and how the header file is identified.	Pre-processor	Link
3.55	16.2	3	The sequence of places searched for the header file specified in quotes in a *#include* "*q-char-sequence*" *new-line* pre-processing directive.	Pre-processor	Link
3.56	16.2	4	The method by which a sequence of pre-processing tokens between < and > or a pair of " characters is combined into a single header name pre-processing token.	Pre-processor	Link
3.57	16.2	5	The mapping between the delimited sequence and the external source file name.	Pre-processor	Link

3.58	16.2	6	The nesting limit to which an #include pre-processing directive may appear due to the *#include* directive of another file.	Pre-processor	Link
3.59	16.6	1	The behaviour of the implementation due to the *#pragma* pre-processing directive.	Pre-processor	Link
3.60	16.8	1	The date/time supplied, as a result of the *__DATE__* macro, if the date of translation is not available.	Pre-processor	Link
3.61	16.8	1	The date/time supplied, as a result of the *__TIME__* macro, if the time of translation is not available.	Pre-processor	Link
3.62	16.8	1	Whether *__STDC__* is predefined and its value.	Pre-processor	Link

6 Indeterminate behaviour

Indeterminate behaviour is defined in the C++ language standard through negative statements. For example, some language statements define that a construct shall not use a particular feature. It is therefore left indeterminate what would happen if such a construct did use a particular feature.

Unique ID	ISO Standard Reference	ISO Standard Paragraph	Description	Classification	
4.01	3.3.1	1	The value used when a variable is used to initialise itself, e.g. *int x = x;*	Initialisation / NameSpace	Link
4.02	5.3.4	15	The value of a POD object created by a new-expression when a new-initializer is omitted.	Initialisation	Link
4.03	5.3.5	4	The value of a pointer that refers to deallocated storage	Pointers	Link
4.04	8.5	9	The value of an object if no initialiser is specified.	Initialisation	Link
4.05	12.6.2	4	The value of a member of a class if it is not otherwise initialised by the constructor.	Initialisation	Link

7 Behaviour that requires no diagnostic

'Behaviour that requires no diagnostic' describes features of the language which do not follow the required or expected rules but for which the language standard states that no diagnostic information is required to be given to the user. Thus it is possible that these issues could be violating the language definition yet no information is passed to the programmer that such a violation has occurred.

Unique ID	ISO Standard Reference	ISO Standard Paragraph	Description	Classification	
5.01	2.7	1	A // comment contains a form feed or vertical-tab character and does not only have white space characters between it and the new-line that terminates the comment.	Lexical Analysis	Link
5.02	2.10	2	Use of an identifier reserved for C++ implementations and standard libraries.	Lexical Analysis	Link
5.03	3.2	3	A program that does not contain exactly one definition for every non-inline function or object that is used in that program.	One Definition Rule	Link
5.04	3.3.6	1 (2)	A name N used in a class S does not refer to the same declaration in its context and when re-evaluated in the completed scope of S.	NameSpace	Link
5.05	3.3.6	1 (3)	If reordering member declarations in a class yields an alternative valid program under certain conditions.	NameSpace / Layout	Link
5.06	3.5	10	If a given object or function can be referred to by values of different type (after all types adjustments)	Type Info	Link
5.07	6.8	3	During parsing, a name in a template parameter is bound differently than it would be bound during a trial parse.	Pre-processor / Template	Link
5.08	7.3.2	4	A namespace-name defined at global scope is also declared as the name of another entity in any global scope of the program.	NameSpace	Link
5.09	10.3	8	A virtual function declared in a class is both defined and declared pure in that class.	Virtual Functions	Link
5.10	12.8	4	Any use of a user defined copy constructor that matches the implicitly declared copy constructor	Function Calls	Link

5.11	14	8	A template that is exported more than once in a program.	NameSpace / Template	Link
5.12	14	8	A non-exported template which is neither defined in every translation unit in which it is implicitly instantiated nor explicitly instantiated in some translation unit	Template	Link
5.13	14.3.3	2	A specialization is not visible at the point of instantiation, and it would have been selected had it been visible.	Template	Link
5.14	14.5.4	1	A partial specialization of a template is not declared before its first use that would cause implicit instantiation in any translation unit.	Template	Link
5.15	14.5.5.1	7	A program contains declarations of function templates that are functionally equivalent but not equivalent.	Template	Link
5.16	14.6	7	No valid specialization can be generated for a template definition, but the template is not instantiated.	Template	Link
5.17	14.6.4.1	7	Two different points of instantiation give a template specialisation different meanings according to the one definition rule.	Template / One Definition Rule	Link
5.18	14.7.3	6	An explicit specialization of a template is not declared before its first use in any translation unit that causes implicit instantiation	Template	Link
5.19	15.4	2	Sets of *type-ids* in exception-specifications in two translation units differ.	Exceptions	Link

8 Conclusions and Recommendations

8.1 Conclusions

8.1.1 One concern that is hampering the use of C++ for safety critical/related applications is a belief in both industry and academia that the dynamic predictability of the (complete) C++ language is not fully understood.

8.1.2 Many of the issues associated with C++ were inherited from its predecessor, C. However, over the years great effort has been put into understanding the holes and pitfalls of C, most notably by Les Hatton [6] who's book "Safer C" laid the foundation for the analysis of the problematical aspects of the language. The current situation with the use of C is that providing a suitably robust fault management strategy is put in place and that verifiable means of compliance to avoiding known problematical language issues is implemented then C has gained a foothold in various safety related applications.

8.1.3 One aspect of any strategy for high-integrity software development should always be the use of a reduced-risk language subset, which formally restricts the use of a language's problematical features. This first requires the problematical features to be identified, and that is what this report has attempted to achieve for C++.

8.1.4 It is anticipated that this report may be used in two ways:

- as guidance to anyone developing a C++ reduced-risk subset, as to the language specification issues that need to be addressed,
- as a bench mark against which any proposed reduced-risk subset can be accessed (again for language specification issues)

8.1.5 In either event, the development of a reduced-risk language subset to address language specification issues is only part of the requirement for high integrity software development. The need to consider avoidance of common programmer errors, clarity of intent to aid maintenance and the development of tool support to police any subset and analyse developed code are outside the scope of this report.

8.2 Recommendations

8.2.1 This report should be given wide circulation, in an attempt to achieve public scrutiny and industrial consensus that the language specification issues that need to be addressed for the safety critical/related use of C++ have been identified.

8.2.2 Any proposed reduced-risk language subset should be assessed against the issues identified here.

9 References

[1] *Guidelines for the use of the C language in vehicle based software*. MISRA. ISBN 0 9524156 9 0

[2] J. Barnes: *High Integrity Ada, The SPARK Approach*, Addison Wesley, 1997.

[3] *C++ Language standard*, ISO/IEC standard 14882, September 1998.

[4] M Hill & E Whiting: *An investigation of the unpredictable features of the C++ language*, QinetiQ Report QINETIQ/KI/TIM/TR043014, May2004

[5] D Reinhardt: *Use of the C++ Programming Language in Safety Critical Systems*, MSc Thesis, University of York, Dept of Computer Science, September 2004

[6] L Hatton: *Safer C, Developing Software for High-Integrity and Safety-Critical Systems*, McGraw-Hill International Series in Software Engineering, 1994

[7] A Koenig*: C Traps and Pitfalls*, Addison Wesley, 1989

[8] *Programming Language C*, ISO/IEC standard 9899:1999(E), November 2001

[9] *Information Technology - Programming Language Ada*, ISO/IEC standard 8652:1995, October 2001

10 Glossary

Accident	An unintended event or sequence of events that can lead to death or serious injury
ALARP	A principal for use in assessing whether safety critical systems are acceptably safe.
Dynamic Predicability	The ability to determine a-priori the run-time values of variables
Error	A departure from the expected or required behaviour of the system through a fault or human error, which could lead to a failure.
Failure	The inability of a system to fulfil its operational requirements which could lead to a hazard
Fault	A defect within a system which may contribute to an error
Fault Management Strategy	A reduced-risk approach that integrates a number of tools and techniques to aid an ALARP approach
Hazard	A situation that occurs from a failure that could lead to an accident
POD	Plain Old Data, essentially a C++ **struct** that would have been legal in C
Safety Case	A reasoned argument, with objective evidence, that a proposed system is acceptably safe for its role and environment.
Safety Critical Software	Software, including firmware, that implements a function or component with the highest safety integrity level requirement (SIL4, as defined by Defence Standard 00-55 & IEC61508).
Safety Related Software	Software, including firmware, used to implement a function or component with some safety requirements, but which is not critical (SIL1 to SIL3 as defined by DS00-55 & IEC61508).
Safety Integrity Level	An indication of the severity of a safety requirement, from SIL1 (minor safety issue) to SIL4 (typically life threatening)
Strong Typing	A programming language (i.e. Ada) where there are tight checks on operations between object so that operations are only allowed on compatible objects
Unambiguous, Self-Consistent and Verifiable	Any reduced-risk language rule subset should be composed of rules that are unambiguous in their restrictions, self-consistent and can be positively verified.
Weak Typing	A programming language where there are loose checks on operations between objects, so that implicit type conversions frequently occur.

Report documentation page

1. Originator's report number:	QINETIQ/S&DU/TIM/CR060019		
2. Originator's Name and Location:	Clive Pygott & Adam Schofield Woodward A109, QinetiQ Malvern		
3. MOD Contract number and period covered:	C/CIP/N03508		
4. MOD Sponsor's Name and Location:	D Wharf		
5. Report Classification and Caveats in use:	6. Date written:	Pagination:	References:
Unclassified/Unlimited	December 2005	vii + 28	9
7a. Report Title:	A Tabulation of the unpredicatable features of the C++ language		
7b. Translation / Conference details (if translation give foreign title / if part of conference then give conference particulars):			
7c. Title classification:	Unlimited		
8. Authors:	Adam Schofield & Clive Pygott		
9. Descriptors / Key words:	SOFTWARE, SAFETY, C++, MISRA		

10a. Abstract. (An abstract should aim to give an informative and concise summary of the report in up to 300 words).

This report analyses the ISO standard for the C++ programming language and identifies those features that are ‘undefined’, ‘compiler dependent’ or in some other way may lead to unpredicatable behaviour. This has been produced as the first step in the process of controlling the use of C++ in safety/security related systems, by identifying the language issues that must be controlled by subsetting the language or analysis of a developed program.

10b. Abstract classification: Unlimited	FORM MEETS DRIC 1000 ISSUE 5

This page is intentionally blank