DRAFT C++ ANNEX OUTLINE

C annex section	Title	Identifier	Status
D.3	Type System	[IHN]	As C, plus…
D.4	Bit Representations	[STR]	As C
D.5	Floating-point Arithmetic	[PLF]	As C
D.6	Enumerator Issues	[CCB]	As C, plus…
D.7	Numeric Conversion Errors	[FLC]	As C
D.8	String Termination	[CJM]	As C, plus…
D.9	Buffer Boundary Violation (Buffer Overflow)	[HCB]	As C, plus…
D.10	Unchecked Array Indexing	[XYZ]	As C, plus…
D.11	Unchecked Array Copying	[XYW]	As C, plus…

D.12	Pointer Casting and Pointer Type Changes	[HFC]	As C
D.13	Pointer Arithmetic	[RVG]	As C
D.14	Null Pointer Dereference	[XYH]	As C
D.15	Dangling Reference to Heap	[XYK]	As C, plus…
D.16	Arithmetic Wrap-around Error	[FIF]	As C, plus…
D.17	Using Shift Operations for Multiplication and Division	[PIK]	As C
D.18	Sign Extension Error	[XZI]	As C
D.19	Choice of Clear Names	[NAI]	As C, plus…
D.20	Dead Store	[WXQ]	As C
D.21	Unused Variable	[YZS]	As C
D.22	Identifier Name Reuse	[YOW]	As C, plus…
D.23	Namespace Issues	[BJL]	Not covered by C
D.24	Initialization of Variables	[LAV]	As C, plus…

D.25	Operator Precedence/Order of Evaluation	[JCW]	As C, plus…
D.26	Side-effects and Order of Evaluation	[SAM]	As C
D.27	Likely Incorrect Expression	[KOA]	As C
D.28	Dead and Deactivated Code	[XYQ]	As C, plus…
D.29	Switch Statements and Static Analysis	[CLL]	As C, plus…
D.30	Demarcation of Control Flow	[EOJ]	As C
D.31	Loop Control Variables	[TEX]	As C, plus…
D.32	Off-by-one Error	[XZH]	As C, plus…
D.33	Structured Programming	[EWD]	As C, plus…
D.34	Passing Parameters and Return Values	[CSJ]	As C, plus…
D.35	Dangling References to Stack Frames	[DCM]	As C

D.36	Subprogram Signature Mismatch	[OTR]	As C, minus…
D.37	Recursion	[GDL]	As C
D.38	Ignored Error Status and Unhandled Exceptions	[OYB]	As C, plus…
D.39	Termination Strategy	[REU]	As C
D.40	Type-breaking Reinterpretation of Data	[AMV]	As C

D.41	Memory Leak	[XYL]	As C, plus…
D.42	Templates and Generics	[SYM]	Not covered by C
D.43	Inheritance	[RIP]	Not covered by C
D.44	Extra Intrinsics	[LRM]	As C
D.45	Argument Passing to Library Functions	[TRJ]	As C
D.46	Inter-language Calling	[DJS]	As C
D.47	Dynamically-linked Code and Self-modifying Code	[NYY]	As C
D.48	Library Signature	[NSQ]	As C
D.49	Unanticipated Exceptions from Library Routines	[HJW]	As C
D.50	Pre-processor Directives	[NMP]	As C, plus…

D.51	Suppression of Language-defined Run-time Checking	[MXB]	As C
D.52	Provision of Inherently Unsafe Operations	[SKL]	As C
D.53	Obscure Language Features	[BRS]	As C, plus…
D.54	Unspecified Behaviour	[BQF]	As C
D.55	Undefined Behaviour	[EWF]	As C, plus…
D.56	Implementation-defined Behaviour	[FAB]	As C
D.57	Deprecated Language Features	[MEM]	As C
D.58	Concurrency – Activation	[CGA]	Not covered by C
D.59	Concurrency – Directed termi	[CGT]	Not covered by C
D.60	Concurrent Data Access	[CGX]	Not covered by C
D.61	Concurrency – Premature Ter	[CGS]	Not covered by C
D.62	Protocol Lock Errors	[CGM]	Not covered by C
D.63	Inadequately Secure Commur	[CGY]	Not covered by C
D.64	Use of unchecked data from a	[EFS]	Not covered by C
D.65	Uncontrolled Format String	[SHL]	Not covered by C

JTC 1/SC 22/WG 23 Document N0563

Additional notes

C++ adds classes, namespaces and an explicit bool type. Vulnerabilities relating to these are addressed later
Type related issues are:
- C++ adds the ability to implicitly create an anonymous class objects if a value is used as a function parameter at a point where a class object is expected (if the value is not itself of the correct class type), and the class has a constructor that can take a single parameter of the value's type. Programmers frequently find this behaviour confusing and it effectively breaks strong typing, so its recommended that it is precluded by using the 'explicit' keyword on any constructor capable of taking a single parameter (other than the copy constructor)
- C++ adds a second use of the keyword 'static' (compared to C). A static class member is a single value, accessible by all objects of that class type. Similarly static member functions are class member functions that don't access any non-static class members. A static member functions can be called without creating a class object. C++ also extends the use of the 'const' keyword, so a const member function does not modify any non-static class members. It is recommended that
- const functions shall not return non-const pointers or references to class members (breaks encapsulation by allowing modification of non-static members)
- member functions that can be static shall be static
    otherwise, if they can be const they shall be const

Not a C++ issue, but missing from the C advice
If you have a switch statement on an enum type with cases for each of the enum members, don't add a default clause with the aim of catching any corrupted values - the compiler is likely to optimise it away

Rather than raw arrays of characters, use the string class from the standard library

The C advice includes checking that array/buffer bounds are respected. One way of ensuring this in C++ is to encapsulate the data array in a class, and only have access via member functions that ensure the legality of indexing - though this raise a strategic issue of what to do if an attempt to access outside the permitted range is detected. In many cases, use of an STL template, such as vector, will provide the required functionality
as [HCB] D.9
as [HCB] D.9

Consider encapsulating the memory to be dynamically allocated in a class, that handles freeing in the class destructor - ensuring that it can only be released once (e.g. have a non-public member pointer for the array, initialised to NULL - reset to NULL if it is necessary to delete allocated memory, and with any allocated memory deleted if not NULL in destructor). This can also help with memory leaks. See [REU] D.39 for ensuring the destructor is called
Not a C++ issue, but not mentioned in the C annex - both C and C++ define different behaviours for signed and unsigned arithmetic. Underflow/Overflow for signed operations is undefined behaviour. For unsigned arithmetic, the behaviour is well defined - but may be unexpected (255+1 == 0 for unsigned char) - so both cases are better avoided

<This item to be dropped>
An additional problem can arise in C++ with objects having similar names appearing in multiple scopes. Where a fully qualified name is used, the intention is clear, but in a member function or after a using statement it may be unclear whether a non-local name refers to an object in file, class or namespace scope. See [YOW] D.22 and [BJL] D.23 for ADL comments

As [NAI] D.19, further confusion can arise where the same name is used in multiple namespace scopes. The rules for deciding which candidate object is to be used are complex and often not well understood, in particular the (implicit) use of Argument Dependant Lookup can cause an object to be selected from a namespace other than the one the programmer expects (banned by MISRA)
See [NAI] & [YOW] D.19/22. In addition, C++ allows anonymous namespaces - which in effect give their members C static linkage. For this reason, an anonymous namespace should not occur in a header file included in multiple translation units. Each inclusion will generate a separate copy of its contents
For class objects, members should be initialised by the class constructor

I think we've added to confusion here by combining operator precedence with order of evaluation. This leads to a statement in the C annex that's just plain wrong "The order of evaluation of the operations in C is clearly defined, as is the order of evaluation". Its operator precedence that is well defined. Order of evaluation is explicitly unspecified - but isn't relevant until you consider side effects [SAM] D.26

The first bullet of the advice in the body of the TR is "The developer should endeavour to remove dead code from an application unless its presence serves a purpose". One of the purposes we mention is as defensive code - however we should point out that compilers may recognise the dead code and optimise it away, so the defensive code the programmer thinks they have may not actually exist - see note on [CCB] D.6

See Notes on [CCB]D.6 and [XYQ] D.28

We should probably say don't use a float as a control loop variable - certainly not with a test for equality

We could recommend using STL container classes, and iterators based on begin()/end() … but any programmer sophisticated enough to do that could probably get the loop right anyway!
Control can be incorrectly transferred into an exception try or catch block by a goto or poorly structured switch statement. Such transfers should be avoided

C++ has passing by pointers or references (syntactically different, but logically almost identical)
Additional advice for pointer/reference parameters - not mentioned by the C annex, but equally relevant for pointers:
- where a value is passed by pointer/reference because its too big to copy efficiently, and there is no intention to modify it, make the parameter const
- consider documenting the intended use of pointer/reference parameters with 'pseudo keywords' such as _IN _OUT or _INOUT (#defined to map to nothing)

The issue of incorrect number of parameters isn't relevant to C++, as use before declaration isn't permitted. However, the advice on limiting the use of functions with ellipsis (…) is still relevant
Probably need to say something about being careful with overloading, and not permitting explicit class construction

C annex advice on the use of errno still applies.
C++ adds an exception mechanism. Some of the issues with exceptions are (see also [REU] D.39):
- (to ensure the correct handler is used) an exception object should not have pointer type
- an exception of class type should be caught by reference
- NULL should not be thrown explicitly (its caught as an int, not a pointer)
- where there are multiple handlers for exceptions of a class type and some of its bases, these should be ordered from most-derived to base (otherwise the wrong handler will be used)
- where there are multiple handlers including a 'catch-all' (...), the catch-all must always be last

A common programming strategy in C++ is to release resource/free memory during the destruction of class objects. If the program exits 'normally', by main returning, then the destructors of all static class objects will be called, and (in getting back to main), as each function is removed from the stack, the destructor of all local class objects also gets called. This is also true if a function exits due to an exception, the stack is unwound from the point the exception is thrown to the point where it is handled. However, there are a number of fault conditions associated with exception handling - such as not having an appropriate handler or throwing an exception in a destructor that is called during the stack unwinding as a previous exception is being handled, that lead to the call of library function terminate(). This closes the program immediately, without unwinding the stack and destroying class objects. This should be avoided. Either:
- Don't use exception handling or libraries that may throw exceptions, or all of the following:
- ensure that every exception explicitly thrown in the program has a handler
- ensure the program has a catch all handler in main
- don't throw exceptions in class destructors
- don't throw exceptions in class constructors, unless it can be shown that no static instances of the class are ever constructed (static construction happens before main, so an exception from the construction of a static class object cannot be caught - note that its only exceptions that propagate from the function that need to be banned. Its OK if they are handled locally)
- generating the object to be thrown should not itself cause an exception to be thrown
- a rethrow statement (throw;) shall only be used in a catch handler

C annex advice still applies.
In C++ encapsulation of allocated data in a class that ensures release on destruction can help mitigate the problem - but see [REU] D.39 for issues that can break this model
 - A class (whether or not it’s a template) may have template member functions. If there is a templated constructor with a single parameter or a templated assignment operator, these hide the default copy constructor and assignment operators respectively. If these default functions are required, they must be provided explicitly
- For clarity, all specializations of a template should be in the same file as the template being specialised
- Also for clarity, whenever a template class is instantiated none of its member functions should be ill-formed (if a function is not actually used in a program, the compiler doesn't have to compile it, so may contain a function applied to the template parameter type that doesn't exist. This is confusing for maintenance. E.g. a generic container class with a sort function that uses the operator >, if this is used to construct a container for a class type that doesn't have operator> defined - if is strictly legal provided the sort function isn't called) E42
C++'s class system allows single and multiple inheritance. It also allows 'virtual inheritance' where a single base class object is shared by multiple derived class objects, e.g. If class D is derived from classes C and B, both of which are derived from A. Then normally, when a class D object is constructed, it will contain two copies of the A object. Under virtual inheritance, the B and C objects is effect share a single A object. This can lead to confusion over member naming, so its recommended that:
- all entities in a multiple inheritance hierarchy same have unique names
- bases classes shall not be both virtual and non-virtual in the same hierarchy

Inheritance also allows the use of virtual functions (polymorphism). Undefined behaviour occurs if a program attempts to make use of a virtual function before the mechanism that supports it is fully established. That is there should be no calls to a virtual member of a class in that class' constructor

<I think C++ uses the same calling strategy as C>

For libraries written in C++, it should be assumed that they may throw exceptions (unless they are explicitly documented as not doing so). See [REU] D.39

Prefer const value declarations to #defines

The C annex argues that it doesn't provide runtime checks, so you cannot turn them off. I don't think C++ has added any (unless some of the RTI behaviour counts - in which case we should say don't suppress RTI where any class has virtual objects - 'cause I'm fairly sure its a compiler option - at least for VS)

C++ is a far larger language than C, so there's both the issue of features that the programmer may be unfamiliar with and the interaction of features that are each inherently complex. The referenced C++ coding standards (MISRA and JSF - as a minimum) bar the use of some features/combinations because experience has shown that, whilst the behaviour of a particular construct may be accurately defined and implemented, programmers don't necessarily understand all the consequences of its use. two specific examples being ADL (Argument Dependant Lookup) and the use of virtual class hierarchies

A specific C++ problem - the need to respect the One Definition Rule (ODR needs explaining here or in an introduction) - this can be assured by:
- any entity used in multiple translation units, shall be declared in a single file
- if such a declaration is in a header file included in multiple translation units, any preprocessor directives that may modify the declaration must be the same at all points of inclusion (e.g. a field of struct cannot #ifdef'ed out in some uses but not others)

Notes from meeting

Put vulnerabilities here. Put language description in .4.

Meta-comment - consider using exceptions to report all errors rather than silence or error values. - place in .4.

add to TR24772-3 6.6.2?

thought - advice, when interfacing through C strings, do not declare and use variables of the C string type; rather use C++ functional equivalents.
JSF reference? MISRA reference.
Vector indexing vs C array indexing. Advice to use X.at(i)

C++ has own casting model and keywords static_cast, … Needs to be evaluated.
Sect 4, C++ has references as well as pointers. Avoids change of notation in indexing structs and pointers to structs. Cannot be overwritten
Consider using a C++ "reference" type instead of pointer if the null value does not need to be representable.

Add to -3 explicitly test for wrap-around??

Vulnerability on unused fields in structs/records and hidden channels? For -1
ditto
Virtual hierarchies introduce issues through declarations of members to derived classes, etc.
Check that overloading of operators is handled.

Change -3 to be correct.
Operator order of evaluation clearly defined but order of evaluation of operands is explicitly undefined. Consider impact of complexity to programmer understanding .

Possible - more thinking needed. Consider removing, and maybe rename as "common programmer errors" and maybe combine with "onscure language features".
Notion of obviscation, volatiles to retain apparent dead code may be nneded.

in -1, add no FP loop counter In - 3, add "in addition to guidance in - 1"

add MISRA C++ recommendations. Consider if this applies to C nested blocks with declarations.

Difference between f(); and
f(void) - document. For C and
C++?

Consider Ada annex for this, and
maybe missing advice in -2.

Dash 1 may need a finalization
vulnerability.

Consider dynamic casts.
Mitigation of casting between
subclasses.

Need a writeup in .4 about how templates work. Consider "concepts". Look at -1 to de-c++ it.

Look to -1 and possibly add cases to the discussion in .3.
Look at hiding in C++ in this context.

declaration of extern functions with parameters. How do prototypes affect this? Any other mechanisms? Mismatch of sources. Check -1 for all issues. Dash 1, calling strategy?

C++ has a self-documenting feature to document exceptions that can be thrown (check?)
Not strictly true. Consider exceptions being thrown.

As C, plus. Can they be suppressed?

Vector indexing vs C array indexing. Advice to use X.at(i) Consider removing, and maybe rename as "common programmer errors" and maybe combine with "likely incorrect expression".

Check.

Check. Recommendation to language definers is to provide the same annex as C provides.

Check.

Check.

Mitigation - use exclusively C++ IO mechanisms (not the only advice).