Document No: WG21 N3733
Date: 2013-08-26
Project: Programming Language C++
References: WG21 N3690, SC22 N4836: ISO/IEC CD14882
Reply to: Barry Hedquist <beh@peren.com>
INCITS/PL22.16 IR

ISO/IEC CD 14882, C++ 2014, National Body Comments

Attached is a complete set of National Body Comments submitted to JTC1 SC22 in response to the SC22 Ballot for ISO/IEC CD 14882, Committee Draft of the revision of ISO/IEC 14882:2011, aka C++ 2014.

This document is a revision to SC22 N4836, CD14882 Collated Comments. The revision contains a consistent numbering scheme for all comments. Comments that contained no numbering were numbered sequentially in the exact order presented in SC22 N4836. Comments that were numbered in the "Line Number" column (column 2) were moved to the MB/NC column (column 1). No other editing was done on any of the comments.

Document numbers referenced in the ballot comments are WG21 documents unless otherwise stated.

Template for comments and secretariat observations

Date:2013-08-26	Document: WG21 N3733	Project: Programming Language C++

MB/ NC[1]	Line number	Clause/ Subclause	Paragraph/ Figure/Table	Type of comment[2]	Comments	Proposed change	Observations of the secretariat
CH 1	all			Ge	The active issues on the issues lists (WG21 N3674, N3682 and N3687) shall be addressed before the standard becomes final.		
CH 2	all			Ge	C++14 is intended to be a bugfix release with minor new features.	Remove any new feature if it negatively affects the quality of the standard.	
CH 3	all			Ge	C++14 is intended to be a bugfix release with minor new features.	Introduce no breaking changes for C++14. This applies specifically to 30.3.1 (~thread()) and 30.6.8 (~future() for asyncs).This also applies to constexpr nonconst member functions, but for this case the CH NB support is not unanimous.	
ES 1				Te	N3674 still includes many unsolved core issues	Solve all the issues identified in N3674.	
ES 2				Te	N3687 still includes many unsolved library issues	Solve all the issues identified in N3687.	
NL1				te	Reconsider adding digit separators, for example as proposed in N3661.		
US 14		(library)		ge	Address open LWG Issues	Appropriate action would include making changes to the CD, identifying an issue as not requiring a change to the CD, or deferring the issue to a later point in time.	
FI 14		[futures]		te	It is unfortunate that futures sometimes block in their destructor and sometimes don't. There have been recommendations to move the futures when unsure, and make sure get() is invoked before the destructor. However, not having a certainly blocking-future in the standard leads to proliferation of custom solutions to the same problem. Similarly, the lack of a certainly-non-blocking future leads to such proliferation.	It seems more future types should be added to establish reasonable semantics. Note that we do not support changing the return type of std::async due to these issues – breaking std::async in any way is harmful to users who already use it for what it was designed, and don't return the futures from it so that there would be confusion about the blocking.	
US 1		All Clauses		ed/ge	In lists of specifications, the use of anonymous bullets makes it difficult (in correspondence and speech) to refer to individual list items. Moreover, the longer the list, the greater the opportunity to mistake the structure, most especially in the presence of bullets in sublists.	In all lists of bulleted items, provide a distinct numbered or lettered identification in place of each bullet. Because paragraphs are already numbered, it seems best to use letters for top-level list items within paragraphs and then to use Roman numerals for any sublist items. (A few parts of the Standard already do this.)	
US 15		All Library		ed/te	Given the adoption of N3655, it is possible to rephrase uses of the type traits throughout and thus both simplify	Replace each occurrence of the form "*cv* typename *typetrait*<…>::type" or the form	

1 **MB** = Member body / **NC** = National Committee (enter the ISO 3166 two-letter country code, e.g. CN for China; comments from the ISO/CS editing unit are identified by **)
2 **Type of comment:** **ge** = general **te** = technical **ed** = editorial

MB/ NC[1]	Line number	Clause/ Subclause	Paragraph/ Figure/Table	Type of comment[2]	Comments	Proposed change	Observations of the secretariat
		Clauses			and clarify the text.	"*cv typetrait*<…>::type" by "*cv typetrait*_t<…>".	
US 4		1.9, 1.10		te	Resolve CWG issues 1441, 1466, 1470 on concurrency. (lower priority).		
US 3		1.9,1.10		te	The current standard accidentally and gratuitously restricts signal handlers much more than was originally intended. Conforming signal handlers cannot even use local variables. They cannot use atomic variables to avoid undefined behaviour as was originally intended.	Correct misstatements, and clarify that atomic<T> operations can be used to communicate with signal handlers, and that objects last modified before a signal handler was installed can be safely examined in a signal handler, e.g. by adopting N3633 or a refinement.	
US 5		1.10, 29.4, 29.6.5		Te	Resolve LWG issue 2075 on concurency.		
FI 1		1-16		te	All Core issues with priorities zero or one up to and including the Core Issues List published in the pre-Chicago mailing shall be resolved	As viewed fit by the Core Working Group	
US 2		1-16		Te/Ge	The active issues identified in WG21 N3539, C++ Standard Core Language Active Issues, must be addressed and appropriate action taken.	Appropriate action would include making changes to the CD, identifying an issue as not requiring a change to the CD, or deferring the issue to a later point in time.	
US 6		2.14		Te	Provide digit separators.	See N3661.	
ES 3		2.14.2		Te	Reconsider adding digit separators for integer decimal literals.	Add digit separators for integer decimal literals as specified in N3661. No counter-example has been presented for integer octal literals.	
ES 4		2.14.2		Te	Add digit separators for integer binary literals.	No interaction has been identified with digit separators for binary literals	
ES 5		2.14.2		Te	Reconsider adding digit separators for integer octal literals	Add digit separators for integer octal literals as specified in N3661. No counter-example has been presented for integer octal literals.	
ES 6		2.14.2		Te	Reconsider adding digit separator for integer hexadecimal literals	A different solution can be evaluated for the conflicting case of digit separators in hexadecimal literals. This case could be solved by using a different prefix to indicate the presence of digit separators.	
ES 7		2.14.2	Table 6	Ed	Header of last columns says:	Modify accordingly table header.	

1 **MB** = Member body / **NC** = National Committee (enter the ISO 3166 two-letter country code, e.g. CN for China; comments from the ISO/CS editing unit are identified by **)
2 **Type of comment:** **ge** = general **te** = technical **ed** = editorial

MB/ NC[1]	Line number	Clause/ Subclause	Paragraph/ Figure/Table	Type of comment[2]	Comments	Proposed change	Observations of the secretariat
					“Octal or hexadecimal constant” This does not include binary constants		
GB 1	Line 40, Page 28	2.14.5	Para 8	Te	The string literal u8"À" (that is, u8"\u00c0") creates a "const char[3]" initialized by { 0xc3, 0x80, 0 }. However, "char" is not guaranteed to be able to represent 0x80.	Change type of u8 string literals to unsigned char, or require signed char to be able to represent 0x80.	
ES 8		3.7.4			Member operator delete[] may take a second parameter indicating the size of the object to be deallocated. However, global operator delete[] does not support this variant.	Provide a global operator delete[] with an optional size parameter along the lines of N3663.	
GB 2	Line 8, Page 78	4.1	Para 2	Te	Reconsider resolution of core issue 616. Under core issue 616, certain lvalue-to-rvalue conversions on uninitialized objects of type unsigned char provide an unspecified value with defined behavior. That is extremely harmful for optimizers, since they must distinguish between a specific unspecified value (which would compare equal to itself, after being copied into another variable) and a fully-uninitialized value.	Further restrict loads of uninitialized unsigned char such that the value can only be stored, and the result of storing it is to make the destination contain an indeterminate value.	
ES 9		5.1.2		Te	Closure objects are never literal types	Consider allowing the generation of literal closure objects.	
GB 3	Line 37, Page 92	5.1.2	Para 11	Te	The access of the non-static data member declared for an init-capture is not specified.	Make the init-capture field unnamed, like other captures.	

GB 4	Line 21, Page 111	5.3.4	Para 8	Te	We are concerned that the change in N3664 may change a small memory leak into a large one.
					Consider
					class P {
					  int x;
					};
					class Q {
					public:
					  Q(){ throw 42; }
					private:		

1 **MB** = Member body / **NC** = National Committee (enter the ISO 3166 two-letter country code, e.g. CN for China; comments from the ISO/CS editing unit are identified by **)
2 **Type of comment:** **ge** = general **te** = technical **ed** = editorial

MB/ NC[1]	Line number	Clause/ Subclause	Paragraph/ Figure/Table	Type of comment[2]	Comments	Proposed change	Observations of the secretariat

					int x[LARGE_NUMBER];
					};
					
					{
					P* p1 = new P();
					Q* q1 = new Q(); // bang :-(
					// don't get here
					delete q1;
					delete p1;
					}
					We fear, if the freedom of N3664 is exercised, that this code block leaks a memory of size at least sizeof(P) + sizeof(Q).
					The C++11 code would only leak the allocation for p1, of size closer to sizeof(P).
					This could result in programs with an insignificant memory leak becoming ones with a more serious leak.		

ES 10		7.6		Te	[[deprecated]] attribute is missing from the CD.	Apply N3394 to the CD.	
US 8		7.6		Te	Paper N3394, "[[deprecated]] attribute," was intended to be included in the CD, but it was unintentionally omitted due to administrative issues.	Incorporate the changes from that paper for the final draft.	
US 10		8.3.4	1	te	The next bullet item appears to the reference the "Size of an object" limit in Annex B. However, in many implementations, object size limits on the stack are quite different from other object size limits, and the limit is very dynamic (especially in the presence of recursion). A check against an fixed (and arbitrary) limit will only cover a subset of the size values that are problematic. In total, we throw on: - negative values and zero (first bullet) - object sizes above the limit We do not throw for: - object sizes which can be allocated successfully - object sizes which cannot be allocated successfully	Do not check at runtime whether the allocated array would exceed the implementation-defined limit on object size.	

1 **MB** = Member body / **NC** = National Committee (enter the ISO 3166 two-letter country code, e.g. CN for China; comments from the ISO/CS editing unit are identified by **)
2 **Type of comment:** **ge** = general **te** = technical **ed** = editorial

MB/ NC[1]	Line number	Clause/ Subclause	Paragraph/ Figure/Table	Type of comment[2]	Comments	Proposed change	Observations of the secretariat
					on the stack, but are less than the object size limit The second item creates significant unpredictability for programmers. Existing VLA implementations for C and C++ lack fully deterministic stack size checks. Obtaining stack is fairly difficult in widely deployed environments (both in terms of availability of the metric and high-performance access to it). An exact check against the dynamic limit is difficult to implement, and would not even cover other causes of stack overflow.		
US 9		8.3.4	1	te	The draft currently requires that if a runtime bound evaluates to 0 at run-time, and exception is thrown. This means that correct C99 code that is also well-formed C++14 code, and has worked fine under the widespread VLA extensions to C++, will fail at runtime; affected code was encountered immediately after the proposal was implemented in G++. A check for negative values makes sense and can be avoided by the programmer by using an unsigned type for the expression. The check against 0 would still be required by the current draft, and is not required by typical VLA usage (because the code deals correctly with this boundary case). It is also surprising because operator new[] lacks such a check. This is a VERY CRITICAL ISSUE..	Allow an array of runtime bound that evaluates to 0 at run-time.	
US 11		8.3.4 [dcl.array], etc.		ed	Two distinct terms of art, *bound* and *extent*, are now used to denote an array's number of elements. For both consistency and improved technical accuracy, a single term of art should be adopted and used throughout the standard.	Because *extent* is the user-visible term used in the Library's interface, its consistent use would avoid breaking existing programs. See the wording proposed in N3549.	
CH 4		8.3.4, 23.3.4		te	VLAs without dynarray is giving wrong direction, and dynarray without full allocator support is just wrong.	Add full allocator support to dynarray or remove both, dynarray and VLAs completely.	
CH 5		8.4.1	p8	te	It's unclear from the text that __func__ is allowed in non function context lambda expressions, i.e., namespace level lambda expressions in initializers.	Specify that __func__ is allowed in such contexts.	
US 12		12.8	31	Te	std::move inhibits copy elision, and so can be a	Ignore calls to std::move, std::move_if_noexcept, and casts to rvalue reference type when	

1 **MB** = Member body / **NC** = National Committee (enter the ISO 3166 two-letter country code, e.g. CN for China; comments from the ISO/CS editing unit are identified by **)
2 **Type of comment:** **ge** = general **te** = technical **ed** = editorial

MB/ NC[1]	Line number	Clause/ Subclause	Paragraph/ Figure/Table	Type of comment[2]	Comments	Proposed change	Observations of the secretariat
					pessimization	determining whether copy elision is permitted	
US 13		12.8	32		Returning a local variable should always imply move semantics.	In return statement, when the expression is the name of a non-volatile automatic object, the expression should be treated as an rvalue for purposes of overload resolution, even if it does not have the same cv-unqualified type as the function return type.	
CH 6		13.5.8	p8	ed	float operator ""E(const char*);// OK should be float operator ""E(const char*);// OK, but reserved (17.6.4.3.5) [usrlit.suffix].	Change the example accordingly.	
FI 2		17-30		te	All Library issues up to and including the Library Issues List published in the pre-Chicago mailing shall be resolved	As viewed fit by the Library Working Group	
GB 5	Line 22, Page 485	20.2.3	Para 1	Ed	The wording describes example code including the call of a move constructor, but there is no requirement stated that T be move constructible.	We would **like** to add a new Para 1 before existing paragraph: Requires: Type T shall be MoveConstructible (Table 20) and MoveAssignable (Table 22). However the MoveAssignable concept currently does not cover cases where the source and destination types may differ.	
ES 11		20.4.2.4	5-6	Te	forward_as_tuple is not currently constexpr	Make forward_as_tuple constexpr.	
CH 7		20.5.1	p2	ed	The example uses the names “index_sequence” and “make_index_sequence” whereas the following sections define “integer_sequence” and “make_integer_sequence”.	Change the names in the example accordingly.	
ES 12		20.6.4		Te	Without operator != users need to evaluate expressions like !(a==b) instead of (a!=b)	Add operator!= for optional<T>	
US 16		20.9.1.3		te	Resolve LWG issue 2118 on unique_ptr.		
ES 13		20.10.11.2		Te	Polymorphic function wrappers do not take move-only callable types in their constructor.	Provide a mechanism to pass move-only callable types to polymorphic function wrappers.	
US 17		20.10.11.2 & 30.6.9		te	Provide a way to pass a packaged_task<T()> to a function accepting function<void()> or another type-erasing callable-wrapper. This is important for concurrency constructs where we	Either change function<> to accept move-only callable types, probably by refcounting the callable, or provide a separate class to turn a move-only callable into a copyable callable.	

1 **MB** = Member body / **NC** = National Committee (enter the ISO 3166 two-letter country code, e.g. CN for China; comments from the ISO/CS editing unit are identified by **)
2 **Type of comment:** **ge** = general **te** = technical **ed** = editorial

MB/ NC[1]	Line number	Clause/ Subclause	Paragraph/ Figure/Table	Type of comment[2]	Comments	Proposed change	Observations of the secretariat
					need to pass tasks between threads using queues. These queues must store a type general enough to represent any task, which includes a task for filling in a future<>. However, function<> currently doesn't accept move-only types like packaged_task<>, so it's not sufficient for the value-type of these queues.		
US 18		20.11.4.3 [meta.unary. prop]	¶ 6	te/ed	The trait is_constructible<T, Args...> is defined in terms of a helper template, create<>, that is identical to std::declval<> except for the latter's noexcept clause.	If the absence of noexcept is critical to this definition, insert a Note of explanation; otherwise, excise create<> and reformulate in terms of declval<> the definition of is_constructible.	
US 19		21.2.3		te	Resolve LWG issue 2232	Proposed Change: Add constexpr to char_traits functions. As a second- best option, resolve LWG issue 2013 to allow libraries to do this as an extension.	
ES 14		21.2.3.1, 21.2.3.2, 21.2.3.3, 21.2.3.4		Te	The following functions are not constexpr in char_traits specializations for char, char16_t, char32_t, and wchar_t: compare() length() find() However, with the addition N3652 a recursive implementation is not needed. Thus they can be easily and efficiently made constexpr.	Make those functions constexpr for the mentioned specializations.	
GB 6	Line 17, Page 689	22.4.1		Ed	17.5.2.3 [objects.within.classes] defines the use of "exposition only" in the library: The declarations for such member objects and the definitions of related member types are followed by a comment that ends with exposition only, 22.4.1 [category.ctype] has members which are preceded (not followed) by a comment ending "exposition only". and 28.12.1 [re.regiter] and 28.12.2 [re.tokiter]	Reformat to follow 17.25.2.3	
GB 7	Line 34, Page 732	23.2.1	Para 4	Ed	Table 98 refers to a and b without defining them. Obviously they are the same as in Tables 96 and 97 but paragraph 23.2.1 / 4 fails to mention Table 98.	Add Table 98 to the scope of paragraph 23.2.1 / 4: In Tables 96, 97 and 98, X denotes ...	

1 **MB** = Member body / **NC** = National Committee (enter the ISO 3166 two-letter country code, e.g. CN for China; comments from the ISO/CS editing unit are identified by **)
2 **Type of comment:** **ge** = general **te** = technical **ed** = editorial

MB/ NC[1]	Line number	Clause/ Subclause	Paragraph/ Figure/Table	Type of comment[2]	Comments	Proposed change	Observations of the secretariat
ES 15		23.2.4	8	Ed	Terminology for table 102 states that “u denotes an identifier”, yet u is not further referred to.	Delete “,u denotes an identifier”.	
ES 16		23.2.4	8	Te	The condition “X::key_compare::is_transparent exists” does not specify that the type be publicly accessible.	Consider the public accessibility of X::key_compare::is_transparent and whether its potential inaccesibility should be banned for a compliant key_compare type.	
GB 8	Line 11, Page 770	23.3.4		Te	The current spec for std::dynarray is contradictory and broken, these open issues should be addressed: - LWG 2253 - LWG 2254 - LWG 2255 - LWG 2264	See related LWG issues at http://cplusplus.github.io/LWG/lwg-active.html	
ES 17		23.4.4.5, 23.4.5.4		Te	Sections are redundant with general associative container requirements at 23.2.4, table 102.	Delete sections.	
ES 18		24.4		Te	Current standard stream does not provide a mechanism for synchronized I/O	Provide a simple mechanism for performing synchronized I/O in multithreaded environments. See N3678	
US 20		Clause 26 [numerics]		ed/te	The Bristol meeting postponed consideration of N3648 because it was assumed that, if adopted, the proposal could be issued in some future Technical Specification. However, N3648 proposes to merge ISO/IEC 29124 into C++14, and it is unclear whether this would even be possible in a TS. Further, such merger is time-sensitive, since ISO/IEC 29124 will be up for review in 2015 and, if merged into C++14, can be retired (“withdrawn”) at that time.	Review and adopt for C++14 the proposal in N3648 (or in a successor document, if any).	
CH 8		26.4		te	Specify user-defined literals for standard complex types.	Accept ISO/IEC JTC1 SC22 WG21 N3660 with the modification to use operator""if for complex.	
US 22		27.4.1	4	Te	Enable standard stream synchronization.	See N3535, N3665, N3678	
GB 9	Line 14, Page 1086	27.9.2	Table 134	Te	C11 no longer defines the dangerous gets() function. Even if we still refer to C99, which includes gets(), it would be preferable to strike std::gets() from <cstdio>	- Remove gets from Table 134 and Table 153. - Add a note to [c.files] saying the C function gets() is not part of C++	

1 **MB** = Member body / **NC** = National Committee (enter the ISO 3166 two-letter country code, e.g. CN for China; comments from the ISO/CS editing unit are identified by **)
2 **Type of comment:** **ge** = general **te** = technical **ed** = editorial

MB/ NC[1]	Line number	Clause/ Subclause	Paragraph/ Figure/Table	Type of comment[2]	Comments	Proposed change	Observations of the secretariat
						- Add the removal of gets to Annex C.3.	
GB 10	Line 14, Page 1103	28.7	Para 12	Te	The current wording is totally broken. Even if the whole proposed resolution at http://www.open-std.org/jtc1/sc22/wg21/docs/lwg-active.html#2018 isn't accepted the "bitwise or" part must be fixed.	Accept the proposed resolution.	
GB 11	Line 4, Page 1128; Line 12, Page 1131	28.12	Para 1 and 2	Ed	17.5.2.3 [objects.within.classes] defines the use of "exposition only" in the library: The declarations for such member objects and the definitions of related member types are followed by a comment that ends with exposition only, 28.12.1 [re.regiter] and 28.12.2 [re.tokiter] have members which are preceded (not followed) by a comment ending "exposition only".	Reformat to follow 17.25.2.3	
US 23		29		Te	Resolve LWG issues 2130, 2138, 2159, 2165 on atomics.		
US 27		30		Te	Resolve LWG issues 2080, 2097, 2100, 2104, 2120, 2135, 2142, 2185, 2186, 2190 on threads.		
US 28		30		Te	Resolve LWG issues 2095, 2098, 2140, 2202 on threads. (lower priority)		
ES 19		30.3.1.3		Te	std::thread destructor calls terminate() if the thread has not been joined. Changing this behaviour is unacceptable for existing code.	A different compatible class or wrapper should be provided to support RAII pattern and joining on destruction.	
US 25		30.3.1.3		te	(Small defect) It is a defect that the thread destructor calls terminate() if the thread has not been joined. Thread is an RAII type and if the user is required to explicitly call .join() or similar in all cases if it has not been called already, this should be done automatically.	A resolution along the lines of that proposed in paper WG21/N3636 or similar would be acceptable.	
US 24		30.6		te	(Severe defect) Like iterators, futures are essential vocabulary types whose major benefit is to permit composability between various providers (containers, async launchers) and consumers (algorithms, async consumers). To be usable as such, they must work predictably. It is a serious defect that ~future and ~shared_future	A resolution along the lines of that proposed in paper WG21/N3637 or similar would be acceptable.	

1 **MB** = Member body / **NC** = National Committee (enter the ISO 3166 two-letter country code, e.g. CN for China; comments from the ISO/CS editing unit are identified by **)
2 **Type of comment:** **ge** = general **te** = technical **ed** = editorial

MB/ NC[1]	Line number	Clause/ Subclause	Paragraph/ Figure/Table	Type of comment[2]	Comments	Proposed change	Observations of the secretariat
					might block unpredictably, depending only on whether the provider was launched with std::async. In all cases in the standard except where the provider is launched with std::async, ~future does not block; if it is launched with std::async, it may block. We understand there are desirable reasons to block (such as to achieve structured resource lifetime control) and not block (such as to achieve responsive nonblocking concurrency), but this decision should be up to each consumer of a given future to select explicitly, not baked inscrutably into an unpredictably dual-mode single future object whose consumer cannot select the appropriate behavior and furthermore the current workarounds to do so are effectively unusable. Futures may or may not block in their destructor, depending on how they were created. Many clients must rely on one behavior or the other, making it impossible to use futures as the general communication mechanism they would like to be.		
GB 12	Line 4, Page 1198	30.6.6	Para 9	Te	Make it explicit that ~future and ~shared_future may block if the future originates from std::async.	Add notes to 30.6.6p9, 30.6.6p10, 30.6.7p11, 30.6.7p12 and 30.6.7p14 after the "releases any shared state" part of the effects saying "[Note: If this is the last reference to the shared state from a call to std::async with a policy of std::launch::async, then this will wait for the async task to complete (30.6.8p5) —End Note]" Add a note to the first bullet of 30.6.4p5: "[Note: this may cause the function that released the shared state to block if this is the last reference to the shared state from a call to std::async with a policy of std::launch::async (30.6.8p5) —End Note]"	
US 26		30.6.8		Te	Deprecate std::async due to the inability to reconcile the blocking semantics of the destructor of the returned values with the growing expected semantics of std::future's destruction. The problems of this inconsistency are outlined in N3630, but the solutions there didn't work. Another solution was proposed in N3637 which also did not satisfy people. Thus, we	Mark std::async as deprecated to help discourage its use and to reconcile the necessity of advising programmers to never pass or return the std::future received from std::async across an interface boundary. Change either 3.6.6p9 to specify that the std::future	

1 **MB** = Member body / **NC** = National Committee (enter the ISO 3166 two-letter country code, e.g. CN for China; comments from the ISO/CS editing unit are identified by **)
2 **Type of comment:** **ge** = general **te** = technical **ed** = editorial

MB/ NC[1]	Line number	Clause/ Subclause	Paragraph/ Figure/Table	Type of comment[2]	Comments	Proposed change	Observations of the secretariat
					request to simply deprecate the problematic feature without changing any behavior in the library, and pave a path forward with new functionality that addresses these concerns.	destructor does not block except when the value is one returned by the deprecated std::async function (or change 3.6.4p5 to specify the equivalent in terms of the shared state).	
FI 15		[basic.life]	paragraph 7	te	See https://groups.google.com/a/isocpp.org/d/msg/std-proposals/93ebFsxCjvQ/myxPG6o_9pkJ It seems that the restrictions of class types with reference members potentially cause a very hard implementation problem. It's palatable to re-fetch pointers and references, but how does one "refresh" a named reference to storage that was destroyed and re-initialized with placement new? In Ivchenkov's example, is it sufficient to destroy the storage_ union and re-initialize the whole union, instead of just its value member?	Clarify what poor programmers need to do if they want to destroy+placement-new-initialize an object of class type, avoiding problems with reference members. Alternatively, consider the solutions presented by Ivchenkov. Our preference leans towards the direction of solutions 5 and 6.	
FI 6		[class.ctor]	paragraph 8	te	In a function returning void, "return E;" where E is of type void is permitted. In contrast, for constructors and destructors, this is not allowed, which is an arbitrary restriction for a corner case.	Remove the prohibition for "return E;" where E is of type void in constructors and destructors.	
CH 9		D.7		te	strstream is dangerous to use and the interface does not fulfill current library requirements.	Delete D.7 from the standard. The CH NB is aware that this proposed change conflicts with the comment to not introduce any breaking changes. So the CH NB support for this comment is not unanimous.	
FI 13		[dcl.attr.grammar]		te	It seems that a [deprecated] attribute fell between the cracks in the EWG->CWG workflow.	Flush the pipeline and add the [deprecated] attribute as proposed in N3394.	
FI 3		[dcl.spec.auto]	paragraph 6	te	As proposed in N3681, an auto specifier should not result in an initializer_list when used with a braced-init-list.	Adopt the solution proposed in N3681, make auto not deduce an initializer_list from a braced-init-list of a single element, make auto with a braced-init-list of multiple elements ill-formed	
FI 4		[dcl.spec.auto]	paragraph 2	te	Function return type deduction also covers conversion functions, that is "operator auto". This is undesirable, because the whole point of a conversion function is to have an explicit (not implicitly deduced) return type. Also, only a single "operator auto" conversion function can exist in a class, limiting its utility.	Exclude conversion functions from return type deduction. Strike conversion-function-id from paragraph 2.	

1 **MB** = Member body / **NC** = National Committee (enter the ISO 3166 two-letter country code, e.g. CN for China; comments from the ISO/CS editing unit are identified by **)
2 **Type of comment:** **ge** = general **te** = technical **ed** = editorial

MB/ NC[1]	Line number	Clause/ Subclause	Paragraph/ Figure/Table	Type of comment[2]	Comments	Proposed change	Observations of the secretariat
FI 5		[dcl.spec.auto]	paragraph 2	te	Function return type deduction avoids the need to repeat the function body to specify the return type in function templates, e.g. the "-> decltype(x1+x2)" below is redundant: template<class T> auto f(T x1, T x2) -> decltype(x1+x2) { return x1+x2; } However, that syntax does not cover exception specifications, again necessitating to repeat the function body: template<class T> auto f(T x1, T x2) noexcept(noexcept(x1+x2)) { return x1+x2; } The specification machinery is readily available with core issue 1351, and the concerns about instantiating definitions to determine properties of the declaration have already been addressed with the introduction of function return type deduction.	Reconsider noexcept(auto), or extend the meaning of "auto" return types to cause exception specification deduction, or find another syntactic means to express deduction of exception specifications.	
FI 8		[expr.prim.lambda]		te	A closure object is not of a literal type, the function call operator of a closure object type is not ever constexpr. These restrictions mean that lambdas cannot be used in constant expression. It seems unfortunate that lambdas and constant expressions do not work together. One of the benefits of relaxing the restrictions of constant expressions was that that relaxation allows writing template code that can be constexpr but is not sub-optimal at run-time and vice versa. It would seem reasonable to allow lambdas to be used in such code.	Allow lambdas to be used in constant expressions, if the captures of the lambda are of literal type, and if the call operator of the closure object type fulfils the requirements for a constant expression otherwise.	
FI 9		[optional.relops]		te	It is unacceptable that optional doesn't have an operator!=.	Define operator!= as the negation of operator==	
FI 10		[optional.relops]		te	It is unacceptable that optional doesn't have operator>, operator<= etc. relational operators in addition to operator<.	Define relational operators as they are defined for tuple and containers. In addition, adopt FI 7 to add a specialization of std::less for optional<T*>.	
FI 7		[pairs.spec], [tuple.special], [container.req		te	std::less is specialized for pointer types so that it yields a total ordering. It seems that utility classes and containers in the library fail to establish the same total ordering, so eg. tuple<T*> or pair<T*, U*> or	Specialize std::less for pair, tuple, optional and containers for pointer types.	

1 **MB** = Member body / **NC** = National Committee (enter the ISO 3166 two-letter country code, e.g. CN for China; comments from the ISO/CS editing unit are identified by **)
2 **Type of comment:** **ge** = general **te** = technical **ed** = editorial

MB/ NC[1]	Line number	Clause/ Subclause	Paragraph/ Figure/Table	Type of comment[2]	Comments	Proposed change	Observations of the secretariat
		uirements.general], [comparisons]			vector<T*> will not have a guaranteed total ordering, since there's no std::less specialization for them and the default std::less will invoke operator< which will use the operator< of the underlying type, hence failing to establish a total ordering.		
FI 16		[support.dynamic]	paragraph 1	te	According to N3396, "In this example, not only is an implementation of C++11 not required to allocate properly-aligned memory for the array, for practical purposes it is very nearly required to do the allocation incorrectly; in any event, it is certainly required to perform the allocation by a process that does not take the specified alignment value into account. This represents a hole in the support for alignment in the language, which really needs to be filled."	Adopt the solution in N3396.	

FI 12		[temp.func.order]		te	In [c++std-ext-14217], Andrew Sutton writes:
					If I have two functions:
					
					template<typename... Args> void f() { } // #1
					template<typename T, typename U> void f() { } // #2
					
					Should overload resolution be able to distinguish these? What I want is this:
					
					f<int, int>() // Calls #2
					f<char>() // Calls #1
					f<int, char, float>() // Calls #1
					
					What I get is, "no matching function" (using an older revision of GCC-4.8). I haven't thoroughly searched the standard for an answer, but I suspect the answer will also be "no".
					If those are template parameters reflect function parameters, then the overloads can be distinguished.
						Make non-deduced function templates with pack arguments less viable than function templates without packs, that is, partially order currently equal/ambiguous candidates so that a pack is a worse match than no pack.	

1 **MB** = Member body / **NC** = National Committee (enter the ISO 3166 two-letter country code, e.g. CN for China; comments from the ISO/CS editing unit are identified by **)
2 **Type of comment:** **ge** = general **te** = technical **ed** = editorial

MB/ NC[1]	Line number	Clause/ Subclause	Paragraph/ Figure/Table	Type of comment[2]	Comments	Proposed change	Observations of the secretariat

					template<typename... Args> void f(Args...);
					template<typename T, typename U> void f(T, U);
					
					It seems like this fact could be extended to non-deduced arguments as well. Just curious.
					
					The question/proposal would seemingly allow metaprogramming techniques that, in conjunction with decltype, allow extracting types from packs without having to resort to traits-like classes with nested typedefs.		

FI 11		[thread.thread.destr]	paragraph 1	te	It is most unfortunate that there is no RAII thread type in the standard. The lack of it leads to proliferation of custom solutions.	We do not support modifying ~thread to join; it has shipped in C++11, and people rely on the terminate() in it. It would be better to introduce a thread_guard that joins the underlying thread automatically upon destruction of the guard.	
US 7		3.7, 5.3, 12.5, 17.6, 18.6, Annex C		te	Enable sized deallocation.	See N3663	
US 21		26.5 [rand], Annex D [depr], etc.		te	The Bristol meeting postponed consideration of N3647 because it was assumed that, if adopted, the proposal could be issued in some future Technical Specification. However, N3647 proposes some deprecations, and it is unclear what it would mean to issue any deprecation in TS form.	Review and adopt for C++14 at least the deprecations proposed by N3647 (or by a successor document, if any). Preferably adopt the entire document, as its proposals are intertwined.	

H:\cc51\Public\SC 22 Project\4850\ISO_IEC 14882_2011_PDAM 1_BSI.doc: Collation successful
H:\cc51\Public\SC 22 Project\4850\ISO_IEC 14882_2011_PDAM 1_NEN.doc: Collation successful
H:\cc51\Public\SC 22 Project\4850\ISO_IEC 14882_2011_PDAM 1_SFS.doc: Collation successful
H:\cc51\Public\SC 22 Project\4850\ISO_IEC 14882_2011_PDAM 1_SNV.doc: Collation successful
H:\cc51\Public\SC 22 Project\4850\ISO_IEC 14882_2011_PDAM 1_ANSI.doc: Collation successful
H:\cc51\Public\SC 22 Project\4850\ISO_IEC 14882_2011_PDAM 1_AENOR.doc: Collation successful
Collation of files was successful. Number of collated files : 6

1 **MB** = Member body / **NC** = National Committee (enter the ISO 3166 two-letter country code, e.g. CN for China; comments from the ISO/CS editing unit are identified by **)
2 **Type of comment:** **ge** = general **te** = technical **ed** = editorial

Template for comments and secretariat observations

Date:2013-08-26	Document: WG21 N3733	Project: Programming Language C++

MB/ NC[1]	Line number	Clause/ Subclause	Paragraph/ Figure/Table	Type of comment[2]	Comments	Proposed change	Observations of the secretariat

SELECTED (number of files): 6 .
FILES IN THIS GROUP(number of files): 6.
PASSED TEST (number of files): 6.
FAILED TEST (number of files): 0.

1 **MB** = Member body / **NC** = National Committee (enter the ISO 3166 two-letter country code, e.g. CN for China; comments from the ISO/CS editing unit are identified by **)
2 **Type of comment:** **ge** = general **te** = technical **ed** = editorial