Document Number: WG21 N4858
Date: 2020-03-20
Reply To: Barry Hedquist
Perennial, Inc.
beh@peren.com

Disposition of Comments
for
CD Ballot, ISO/IEC CD 14882

Template for comments and secretariat observations

Date:2020-02-15	Document:	Project: 14882

MB/ NC[1]	Line number	Clause/ Subclause	Paragraph/ Figure/Table	Type of comment[2]	Comments	Proposed change	Observations of the secretariat
FR 001				ge	There have been many papers in the last few months exploring the coroutines design space including P1745, P1713, P1662, P1663, P1344. These evolutions seems worth pursuing, notably because coroutines cannot be used in environments where heap allocations are impossible. However, it is not clear that there will be a reasonable non-breaking path towards these evolutions as they almost all assume a separation of coroutine_handle and suspend points (as described in P1745), which would be a breaking change if adopted after the publication of the C++20 standard.	Ensure that the final specification of coroutines do not preclude further extensions including those outlined in the paper P1745.	Rejected There was no consensus to adopt this change.
DE 002					P1020R1 ((https://wg21.link/p1020r1) proposed some helper functions for smart pointers that do not necessarily initialize values. Providing better performance but bring more risk. However, the technical term "default initialization" used in their name does not always initialize (as value initialization does). Therefore, it is highly confusing for programmers to call an ordinary function make_unique_default_init() make_shared_default_init() allocate_shared_default_init() If you ask what is safer: make_shared() or make_shared_default_init() almost every programmer says the latter. For this reason, the more risky version should have a name that demonstrates the risk rather than giving a false sense of safety.	We strongly request to rename them to make_unique_nonvalue_init() make_shared_nonvalue_init() allocate_shared_nonvalue_init() (with or without a _ between non and value).	Accepted with Modification See P1973
DE 003					C++ has a key problem being far too complicated for beginners. Now we make it even worse. Instead of size(), which is self-explanatory but	We request to use the name count() or at least remove ssize() otherwise.	Rejected There was no consensus to adopt this change.

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
					creates warnings, we introduce ssize(), which avoids the warnings but is way harder to teach and NOT self-explanatory. There was a nice alternative proposal: count() as proposed in https://wg21.link/p1764. This is both self-explanatory and avoids the confusing warnings.		
CZ 004				ge	With introducing of P1152 (Deprecating volatile) we deprecated most usages of volatile keyword. We should reflect the changes in library by adopting the changes there.	Adopt P1831 (deprecating volatile: library)	Accepted See P1831
CZ 005				ge	Please consider adopting P0593 (Implicit creation of objects for low-level object manipulation) to fix long-standing problems with low level object creation and UB.	Adopt P0593 (Implicit creation of objects for low-level object manipulation)	Accepted See P0593
RU 006		[atomics.types.operations]	Paragraph 2	te	Fix default initialization of C++ atomic variables to fit user expectations. Default constructor of std::atomic has to initialize the atomic object with value of T{}. Default constructor of std::atomic_flag has to initialize to clear state.	Apply wording from P0883.	Accepted See P0883
RU 007		[basic.life]	Paragraph 8, bullet 3	te	In many cases it is impossible to use a pointer returned from placement new or std::launder (for example in std::vector, std::variant, std::optional, std::uninitialized_*+std::destroy*). Because of that issue all the standard libraries have undefined behaviours in widely used types. The only way to fix that issue is to adjust the lifetime rules to auto-launder the placement new. Dropping the “const“ and “reference” requirement from paragraph 8 [basic.life] removes UB from std::vector and std::optional. Additional removing of the “potentially-overlapping” requirement removes UB from std::variant.	Apply the following changes to the [basic.life] paragraph 8 bullet 3: – the type of the original object is not const-qualified~~, and, if a class type, does not contain any non-static data member whose type is const-qualified or a reference type~~, and Optionally remove the to the bullet 4 from [basic.life] paragraph 8: ~~– neither the original object nor the new object is a potentially-overlapping subobject ([intro.object])~~.	Accepted with Modification See P1971
RU 008		[class.cdtor]	Paragraph 2	te	Clarify that *this in constructors could not alias with reference input parameters of the same type. This is useful for optimizing all the constructors of all the classes.	Apply the following changes to the [class.cdtor] paragraph 2: During the construction of an object, if the value of the object or any of its subobjects is accessed	Rejected There was no consensus to adopt this change.

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
						through a glvalue that is not obtained, directly or indirectly, from the constructor's this pointer, the value of the object or subobject thus obtained is ~~unspecified~~undefined.	
RU 009		[compliance]		te	Make <compare> freestanding.	Apply wording from P1855.	Accepted See P1855
FI 010		[compliance]			<coroutine> uses <compare>, <compare> is not freestanding	Adopt P1855R0	Accepted with Modification See P1855
RU 011		[dcl.fct.def.general]	Paragraph 8	te	Make __func__ usable in constant expressions. Resolve CWG 2362 (http://www.open-std.org/jtc1/sc22/wg21/docs/cwg_active.html#2362)	Change the definition of __func__ to a static constexpr variable: static constexpr char __func__[] = "function-name";	Rejected There was no consensus to adopt this change.
RU 012		[expr.prim.req]	Paragraph 6	te	Accessing an incomplete type in requires-expression results in substitution failure. This is a dangerous approach that provokes odr-violations. Making the program ill-formed is a better approach that protects from hard detectable odr-violations.	Add to the [expr.prim.req] paragraph 6: In such cases, the requires-expression evaluates to false; it does not cause the program to be ill-formed if the requires-expression does not refer to incomplete type.	Rejected There was no consensus to adopt this change.

RU 013		[string.cons]	Paragraph 30	te	Because of the implicit conversion of arithmetic types it is error prone to use the basic_string::operator=(charT c):
					
					double d = 3.14;
					std::string s;
					s = d; // Compiles
					
					Make sure that the program is ill-formed if an implicit conversion from arithmetic type happens while assigning to std::basic_string. Or at least make sure that the program is ill-formed if an implicit conversion from floating point type happens while assigning to std::basic_string.
						Apply the following changes to the [string.cons] paragraph 30:
						template <class T>
						constexpr basic_string& operator=(~~char~~T c);
						Constraints:
						is_same_v<T, charT> is true
						Effects: Equivalent to:
						return *this = basic_string_view<charT, traits>(addressof(c), 1);
						
						Change the [basic.string] synopsys:
						template <class T>
						constexpr basic_string& operator=(~~char~~T c);
							Rejected
							There was no consensus to adopt this change.

RU		[string.view.		te	Forbid assigning an rvalue std::basic_string to std::basic_string_view. It will allow to detect errors	Add to the [string.view.template]	Rejected

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
014		template]			at compile time, rather than at runtime. Resolve LWG 3068 (https://cplusplus.github.io/LWG/issue3068)	basic_string_view synopsis: template <class A> basic_string_view& operator=(const basic_string<charT, traits, A>&&) = delete;	There was no consensus to adopt this change.
FI 015		[support.limits.general]			Various feature-testing macros are missing; examples are ones for span and concepts.	Discuss, modify and adopt Barry Revzin's revision of SD-6, and adopt the resulting changes into the IS working draft.	Accepted with Modification See P1902
RU 016		[temp.alias]		te	Fix pack expansion into fixed alias template parameter list by resolving CWG 1430 (http://www.open-std.org/jtc1/sc22/wg21/docs/cwg_active.html#1430). Desired outcome is to allow the following code to compile: #include <type_traits> template <typename... Xs> auto f (Xs...) -> std::invoke_result_t<Xs...>;	Resolve CWG 1430.	Rejected There is no consensus to adopt this chage at this time. However, this issue will be given consideration for a future resolution post C++20.
GB 017		01		Te	This is an explicit request to address all open issues on the core and library list.	Address Core and Library open issues.	Accepted
DE 018		01.02 31.8 31.8.1 31.8.2 31.8.3 31.8.4 31.9 31.10			P0883: Fixing Atomic Initialization (https://wg21.link/p0883) went through SG1 and LEWG to be adopted for C++17. This is an important fix for the broken atomic initialization.	Apply this fix through LWG. According to N4830 this paper now applies to: 31.2 Header <atomic> synopsis [atomics.syn] 31.8 Class template atomic [atomics.types.generic] 31.8.1 Operations on atomic types [atomics.types.operations]: 31.8.2 Specializations for integers [atomics.types.int] 31.8.3 Specializations for floating-point types [atomics.types.float] 31.8.4 Partial specialization for pointers [atomics.types.pointer] 31.9 Non-member functions [atomics.nonmembers] 31.10 Flag type and operations [atomics.flag]	Accepted with Modification See P0883

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
US 019		02	p1	TE	We normatively refer to ISO/IEC 9899:2011 which is not the latest version of the C Standard.	We should be tracking against ISO/IEC 9899:2018 (aka, C17) instead. C17 is near-identical to C11 in that it was a bugfix release that only resolved DRs, with one exception (ATOMIC_VAR_INIT was deprecated in C17)	Accepted See P1971
US 020		02	p1	TE	We normatively refer to ISO/IEC 9899:2011 which is not the latest version of the C Standard.	We should be tracking against ISO/IEC 9899:2018 (aka, C17) instead. C17 is near-identical to C11 in that it was a bugfix release that only resolved DRs, with one exception (ATOMIC_VAR_INIT was deprecated in C17)	Accepted See P1971
US 021		03.06 [denfs.block]		Ed	[intro.defs] defines 'block' only as a verb (3.6 [defns.block]), for the memory model. We also use 'block' to define a brace-enclosed region of code, per 8.3 [stmt.block]. The index entry for 'block' refers to the [defns.block] definition, but has sub-entries that apply only to the [stmt.block] definition.	Add a second definition for block as a brace-enclosed region of code, and update misdirected index references to the new definition. Index entries for 'block' will want to clarify between the verb and the noun.	Accepted - Editorial
GB 022		04.01		Ed	Allow access to standard library names via import [intro.compliance] 4.1 p5 "The names defined in the library have namespace scope (9.7). A C++ translation unit (5.2) obtains access to these names by including the appropriate standard library header (15.2)."	Add that standard library names can also be accessed by importing the appropriate header unit.: … library header (15.2)<ins> or importing the appropriate standard library named header unit (10.3), [headers] 16.5.1.2 p4</ins>.	Accepted - Editorial
JP1 023		04.02	p5	ed	There are also many "[Note 1 to entry:" style notes in the document as well as "[Note:". They should also be referred to.	Add "[Note 1 to entry:" and terminated by "--end note]" at the end of the 2nd sentence appropriately.	Accepted - Editorial
US 024		05.02 [lex.phases]	2	Te	Undefined behavior lexing the program has no place in a modern standard, and this should be a diagnosable error. This concerns the following text: "Except for splices reverted in a raw string literal, if a splice results in a character sequence that matches the syntax of a *universal-character-name*, the behavior is undefined." The behavior should either be well defined, and produce the universal character; ill-formed (diagnostic required); or conditionally supported as forming the universal character (must be documented and diagnosed if not supported)	Make this either a diagnosable error or produce the universal character (potentially as conditionally supported behaviour).	Rejected There was no consensus to adopt this change.
US 025		05.02 [lex.phases]	4	Te	Undefined behavior lexing the program has no	Make this either a diagnosable error or produce	Rejected

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
					place in a modern standard, and this should be a diagnosable error. This concerns the following text: "If a character sequence that matches the syntax of a *universal-character-name* is produced by token concatenation (15.5.3), the behavior is undefined." The behavior should either be well defined, and produce the universal character; ill-formed (diagnostic required); or conditionally supported as forming the universal character (must be documented and diagnosed if not supported)	the universal character (potentially as conditionally supported behaviour).	There was no consensus to adopt this change.
US 026		05.04 [lex.pptoken], 6.5 [basic.link], 10 [module.unit], 10.4 [module.global], 15 [cpp], 15.4 [cpp.glob.frag]	See P1857r0	te	import is a directive while module is not. This makes it difficult to do dependency scanning via partial preprocessing, which was the design intent of making import a directive.	Make module a directive. See P1857r0 for details.	Accepted with Modification See P1857
US 027		05.04 [lex.pptoken]	4	Te	It is undefined behavior to have an unmatched ' or " character as a single-character token (i.e., surround by whitespace). Undefined behavior lexing the program has no place in a modern standard, and this should be a diagnosable error.	Make this a diagnosable error.	Rejected There was no consensus to adopt this change.

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
US 028		05.06 [lex.token]		Te	A token is defined as either an *identifier*, a *keyword*, a *literal*, an *operator*, or a *punctuator*. While the first 4 all have definitions for their grammar term, there is no definition of a *punctuator*.	Add the missing definition for the grammar term *punctuator*. It probably should be "one of:" the set of tokens in *preprocessing-op-or-punc* that are not also present in the grammar production for *operator*	Accepted with Modification See P2103
NL 029	1	05.10 [lex.name]	[tab:lex.name.allowed]	te	Allowed characters include those from U+200b until U+206x; these are zero-width and control characters that lead to impossible to type names, indistinguishable names and unusable code & compile errors (such as those accidentally including RTL modifiers).	Disallow invisible characters in this range	REJECTED There was no consensus to adopt this change.
JP2 030		05.13.04		ed	At binary-exponent-part, a small p and a large P are mixed and hard to distinguish with current font.	Use another font which has distinct typefaces for p and P.	Rejected There was no consensus to adopt this change.
US 031		06.02 [basic.def.odr]	09.2.2	Ed	The last line of the example is (mildly) confusing: { [n]{ return n; }; }; *// OK* Yes 'n' is reachable but the code will not compile.	I would suggest changing the line to: { [n]{ return n != 0; }; }; *// OK*	Rejected There was no consensus to adopt this change.

GB 032		06.04.2		Ed	[basic.lookup.argdep] example paragraph 5 lacks necessary namespace qualification
					In section 6.4.2 paragraph 5:
					Translation unit 1:
					namespace R {
					export struct X {};
					export void f(X);
					}
					namespace S {
					export void f(X, X);
					}
					the f() declaration in S in translation unit 1 lacks namespace qualification for X.
						Insert R:: as shown:
						
						namespace R {
						export struct X {};
						export void f(X);
						}
						namespace S {
						export void f(<ins>R::</ins>X, <ins>R::</ins>X);
						}
							Accepted - Editorial

US 033		06.05		te	Import declarations may only appear at the top level, outside of any bracing. This means they cannot appear inside a linkage block (9.10) 'extern "C" { … }'. Header translation (15.2/7) is	Permit import-declarations within linkage blocks at the global namespace level. There is a commonality with the next issue, which is noted as an editorial issue below.FYI both Clang's	Accepted with Modification See P2103

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
					specified as a rewrite of a #include directive into an import declaration. Unfortunately header files exist that contain (different sets of) #includes (of C headers) inside linkage blocks – even when the included header is itself C++-aware and encapsulates its contents in a linkage block. Such included headers may be included elsewhere from the top level and be suitable for include translation. It would be unfortunate if the decision as to whether include translation should be performed had to be made context-sensitive, examining previously lexed tokens. It could also be surprising to users. Thus, in practice, implementors must permit import declarations inside linkage blocks (at least for these translated cases).	module extension and GCC's C++ Modules found it necessary to implement this extension, due to glibc header files.	
US 034		06.05		te	Import declarations may only appear at the top level, outside of any bracing. This means they cannot appear inside an export block (10.2) 'export { … }'. Such non-generality of export blocks can lead to user confusion. The original intent was to simplify module preamble parsing, along with the backtracking the original ATOM design envisioned. In the p1103 specification it simplifies the recognition of header imports during phase 4, where their exported macros must be made available. However, as specified in the previous issue, implementors must account for some cases occurring within linkage blocks. Permitting imports within export blocks would not further inconvenience implementations. Note, relaxing this restriction does not break the status quo with lexing import declarations as preprocessing directives. The export semantics are applied at the c++-parser level.	Permit import-declarations within export blocks at the global namespace level. There is a commonality with the next issue, which is noted as an editorial issue below. In a named module, an export block of imports would be restricted to the beginning of module purview, and perhaps only permitted to contain import declarations, to retain the current requirement of placing import declarations immediately after the module declaration.	Rejected There was no consensus to adopt this change.
US 035		06.05		te	The CD requires implementation linkage promotion for internal-linkage entities that are referenced from locations made visible to other TUs. For example:	P1498 proposes making such references ill-formed. Wording should be completed to specify that. A decision should be made as to whether the error is indicated when compiling the module interface that exposes the internal entity, or	Accepted with Modification See P1815

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
					export module foo; static void internal () {} export template<typename T> void widget () { internal (); } instantiations of 'widget' will need to reference 'internal', thus its symbol must be globally visible. This presents implementation difficulties and inhibits certain optimizations. Papers p1347 and p1395 describe the issue in some detail.	whether referencing that internal entity from a different TU (instantiating a template, calling an inline function, etc).	
US 036		06.05		te	The semantics of the private module fragment (PMF) are underspecified. It appears well formed to declare an entity in the interface purview and define it in the PMF. How does this interact with inlining, instantiation and internal linkage definitions? The intent of the PMF is as-if it is a separate module implementation unit, but we do not define the boundary between the interface purview and the PMF as a translation unit boundary. Implementations may defer instantiation (of function definitions) to the end of translation. Similarly internal linkage and inline functions (that are ODR-used) must be defined at the end of translation. Perhaps emission of Compiled Module Interface (CMI) should be deferred to the end of translation – and not at the beginning of the PMF. It will therefore observe entities declared or defined in the PMF. This may create implementation difficulty to preserve the semantics of the PMF being as-if a separate translation unit. Alternatively, the boundary could be specified as a new kind of 'end of translation' point. This seems a dramatic change.	Remove the private module fragment. It's semantics are too ill-defined, and I do not believe there is sufficient experience to define them at this point.	Rejected There was no consensus to adopt this change.
US 037		06.05 & 10.2		ed	Linkage and export blocks are the only two namespace-scope braced constructs that do not introduce a scope. As described above, implementations must at least permit some imports within linkage blocks. This leads to the confusing situation that imports are neither a top-level construct, nor one that is permitted at any global scope. If both the above restrictions are relaxed, import-declarations can be described simply as entities appearing at global scope.	Accept the previous two changes.	Rejected There was no consensus to adopt this change. .

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
CA 038		06.05 [basic.link], 9.8 [namespace .udecl], 9.10 [dcl.link], 12.1 [over.load]		te	Trailing *requires-clause*s are ignored in various places covering the identity of functions.	Update appearances of “parameter-type-list” in [basic.link], [namespace.udecl], [dcl.link], and [over.load] to also take *require-clause*s into account.	Accepted with Modification See P1971
FR 039		06.05.2/4		te	The current wording allows a non-exported function to be visible to an importing translation unit via ADL. That is too broad and goes against the abstraction boundaries that modules are supposed to bring.	Specify that these names are available only during the second phase of ADL during template instantiation and only if the module interface unit is on the path of instantiation.	Accepted - Editorial
US 040		06.06.2 [intro.object]	n/a	te	The core language changes from P0593R5 to enable implicit object creation are important to support P1004R2. These changes should be viewed as resolving a defect in C++.	Apply the changes from P0593R5, other than the addition of new standard library functions, and treat them as the resolution of a defect report.	Accepted See P0593 Must load latest paper

US 041		06.06.3 [basic.life]	8	te	The subject paragraph is missing wording which disallows using a glvalue which referred to a member subobject from being used to refer to an object that is not similarly such a member subobject. This means that a subobject of one class might become a subobject of an altogether different class.
					For example, with the wording as it is, the following should return typeid(Z) if the assertions pass:
					#include <new>
					#include <memory>
					#include <cassert>
					#include <typeinfo>
					struct X { int x; };
					struct Y : X {
					virtual ~Y() = default;
					};
						Add a new bullets to 6.6.3 [basic.life] paragraph 8:
						• the respective complete objects (6.6.2 [intro.object]) of the original object and the new object are either the same or meet these requirements in turn, and
						• the original object and the new object are each the same subobject in relation to their respective complete objects as each other.
							Accepted with Modification
							See P2103

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

					struct Z : Y { };
					const std::type_info &f() {
					  Y *yp = new Y;
					  int &yx = yp->x;
					  assert(sizeof(Y) == sizeof(Z) &&
					  alignof(Y) == alignof(Z));
					  Z *zp = new ((void *)yp) Z;
					  assert(std::addressof(yx) ==
					  std::addressof(zp->x));
					  return typeid(*static_cast<Y *>(
					  reinterpret_cast<X *>(&yx)));
					}
					It is surprising that the set of types associated with the most derived objects of the objects that are pointer interconvertible with the pointee of a pointer value may change between points at which the pointer value may be plainly dereferenced.
					In this specific case, yx was obtained as a reference to an int subobject of an object whose most derived type is Y. Yet yx came to refer to a subobject that was not of any object whose most derived type is Y. yx should not be allowed to change identities in this manner.
					This appears to be a defect in all prior versions of C++.		

US 042		06.06.3 [basic.life]	8	te	The restriction on automatically referring to the new object when a class type contains a non-static data member of const-qualified or reference type renders use of the pointer returned by data() of a std::vector of such class types unsafe without applying std::launder. In particular, if a pointer returned by data() on a one-element vector of a class type that is subject to the restriction is stored and then followed by a pop_back() and a call to push_back, then the stored pointer would not be usable to access the new element without applying launder. The need to apply launder is not currently indicated by the library wording and is presumably unwanted.	In the bullet, strike: • the type of the original object is not const-qualified~~, and, if a class type, does not contain any non-static data member whose type is const-qualified or a reference type~~, and Remove, from subclause 6.6.2 [intro.object] paragraph 2, the note beginning with "[i]f the subobject contains a reference member or […]" and the associated example.	Accepted with Modification See P1971

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

US 043		06.08 [basic.def] 9 [dcl.dcl] 9.1.8.3 [dcl.type. elab] 9.6.2 [enum.udecl] 9.8 [namespace .udecl] 11.3 [class.mem] 15.10 [cpp.predefi ned]	02.17
			
			1
			
			Table 17
				te	The syntax of the using enum feature is very subtle, which may be confusing to users. Consider, for example:
					struct B {
					protected:
					enum class E { e1, e2 };
					};
					
					class D1 : public B {
					public:
					using B::E;
					// Creates a public member using
					// declaration in 'D1' for 'B::E'.
					};
					
					class D2 : public B {
					public:
					using enum B::E;
					// Creates a public member using
					// declarations in 'D2' for
					// 'E::e1' and 'E::e2'.
					};
					
					void f() {
					D1::E v1; // Ok
					D2::E v2; // COMPILE FAILURE
					D1::e1; // COMPILE FAILURE
					D2::e2; // Ok
					}
					Additionally, the syntax looks similar to elaborated type specifiers, which may also lead to confusion. Please reconsider this feature; it was added very late in the C++20 design process, and it may be better to move it to C++23 to give us more time to work on the design.
						Possible resolutions include:
						• Disallow class member using enum declarations.
						• Move using enum from C++20 to C++23 to give us additional time to work on it.
							Rejected
							There was no consensus to adopt this change.

CZ 044		07	07.7	te	Small Buffer Optimization is a common and widespread container implementation strategy for containers that do not have as strong a guarantee for iterator invalidation as std::vector, such as std::string. With both vector and string being made constexpr, Small Buffer Optimization cannot be done legally at constexpr time due to	Change [expr.const/5.1-2] to read as follows: — for a call to std::construct_- at or std::ranges::construct_at, the first argument, of type T*, does not point to either storage allocated with std::allocator<T> or an automatic storage duration variable of type T, or the	Accepted with Modification See P1971

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

					the inability to invoke std::construct_at or std::destroy_at on memory that has automatic storage duration. That is, the wording does not seem to bless the following program as legal:
					
					`#include <memory>`
					
					`constexpr int foo () {`
					`int x = 0;`
					`std::destroy_at(&x);`
					`std::construct_at(&x, 2);`
					`return x;`
					`}`
					
					`constexpr int bar () {`
					`int x[2]{};`
					`std::destroy_at(&x[0]);`
					`std::construct_at(&x[0], 3);`
					`return x[0];`
					`}`
					
					`int main () {`
					`static_assert(foo() == 2);`
					`static_assert(bar() == 3);`
					`}`
					
					This presents problems for a future wherein someone wishes to work with SBO containers, and may present a serious problem for the future work with getting constexpr-capable containers to escape compile-time and serialize into something that can be used at runtime.
						evaluation of the underlying constructor call is not a core constant expression, or
						— for a call to std::destroy_-
						at or std::ranges::destroy_at, the first argument, of type T*, does not point to either storage allocated with std::allocator<T> or an automatic storage duration variable of type T, or the evaluation of the underlying destructor call is not a core constant expression.	

JP3 045		07.02.2	p3.6	ed	"C1 is reference-related to C1" looks like a typo.	C1 is reference-related to C2	Accepted - Editorial
GB 046		07.05.4		Te	Implementations should be allowed to cache the results of concept specialisation evaluations The current wording of [expr.prim.id]/4 precludes the possibility of caching concept specialisations (which has led to demonstrable performance issues in implementations that are standard-conforming contrasted against implementations	Add wording to the effect of: A program which causes a given set of template arguments to change their satisfaction of a constraint, and relies upon this change, is ill-formed, no diagnostic is required.	Accepted with Modification See P2104

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
					that cache results).		
US 047		07.05.4.1	2	ed	If the entity is a template parameter object for a template parameter of type T (13.1), the type of the expression is const T." follows from [temp.param]/1.6 (which establishes that the template parameter object has that type).	Strike the sentence.	Accepted - Editorial
BG7 048	P 117-119	07.06.2.3		ge	The same comments as 6, this time for the Awaitable concept. await_ready, await_suspend, await_resume are too uniformly named, with no differentiation b/w the observer await_ready and the other functions.	Rename await_suspend to await_on_suspend and await_resume to await_on_resume.	Rejected There was no consensus to adopt this change.
BG2 049	P 118	07.06.2.3	(3.7)	ge	The variants of await_suspend() that return void and bool can introduce unexpected stack frames. Because of this coroutine authors cannot rely that co_await will not introduce stack frames.	Leave only the await_suspend() variant that returns a coroutine_handle.	Rejected There was no consensus to adopt this change.
BG1 050	P 117-119, P 134-135, P 150, P 204-206, P 520-524, etc.	07.06.2.3, 7.6.17, 8.6.4, 9.4.4, 17.12, etc.		ge	The coroutines design that was adopted has serious flaws: - The language-provided constructs cannot be used without helper types (“coroutine types” and “awaiters”). It is expected that these helper types will be provided by library writers. We believe that major and subtle differences will often make it hard or impossible to compose helper classes written by different authors. - The semantics of the co_await operator enforces a “trampoline” style transfer of control and hinders the adoption of a tail-call-style transfer of control in the future. - The coroutine_handle::resume() method is untyped which makes transfer of control type-unsafe. - Furthermore, the untyped resume operation separates the transfer of control from the transfer of data. This affects performance, jeopardizes safety	Delay coroutines from C++ 20 to C++ 23 and work actively to reach a qualitatively better design and substantial usage experience beyond the current limited set of users and use cases.	Rejected There was no consensus to adopt this change.

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
					and increases the risk of bugs. - There can be multiple copies of the same coroutine_handle raising the risk of inadvertent incorrect resumption of a coroutine. - The asymmetric variants of await_suspend() cause calls between coroutines to accumulate stack frames. This is contrary to the very definition of coroutines and precludes certain synchronous usage patterns. - The symmetric variant of await_suspend() is forced to use a trampoline which presents a challenge to the optimizer. - Having both the symmetric and asymmetric variants of await_suspend() means that coroutine authors cannot rely that using co_await will prevent unbounded accumulation of stack frames. - The type-erased coroutine frame introduces allocations that are out of the control of the user. The proposed solution (to declare an unimplemented override of new for the coroutine frame that causes compilation errors when the compiler fails to elide the allocations) ensures only the detection of these allocations but doesn't give a recipe for eliminating them. Additionally, after the TS was merged into the WD, early adopters identified problems (in P1662R and P1745R0) in the design related to cancellation and cleanup which the Bulgarian NB finds valid. The authors have indicated that eliminating these issues will require breaking changes to the current design. We strongly support resolving these problems without		

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
					breaking changes. More importantly, these findings suggest that although the TS has been used in production, the usage experience has been limited to just a few specific use cases. Last but not least, at least three promising alternatives have been presented in the last two years - P1063, P1430 and P1745 - and for the most part these alternative designs cannot be reached from the current design without breaking changes. The Bulgarian NB is seriously worried that the work in the design space is not mature and that important design directions have not been considered.		
GB 051		08.05.4		Te	Range-based for loops should look for ranges::begin and ranges::end ranges::begin and ranges::end perform similar work to the range-based for loop. It would be good to have these as a bullet in the lookup for what a range-for looks for to help with the deprecation process of std::begin and std::end in a future standard.	Add a sub-item that allows begin-expr to be determined from a call to ranges::begin(range) and end-expr to be determined from a call to ranges::end(range). Consider adding this sub-item between 1.3.1 and 1.3.2.	Rejected There was no consensus to adopt this change.
US 052		08.06.4	1	ed	'A coroutine shall not return to its caller or resumer by a return statement (8.6.3)' suggests this is a dynamic restriction – i.e. a coroutine may contain return-statements, provided they are not reached during execution. This is believed a wording error.	Replace with 'A coroutine shall not contain a return statement'. I believe it clear that 'contain' already includes unreachable statements of constexpr-ifs. Thus one cannot use constexpr-ifs to define a function that is conditionally a coroutine.	Accepted with Modification See P1971
US 053		08.06.4 [stmt.return. coroutine]	3	TE	Apply remaining coroutine TS issues to the working paper.	Apply resolution of the issue 34 "Mandate the return type for return_void and return_value to be void." From p0664r8	Accepted with Modification See P1971
GB 054		09.01.5		Te	P1331R2 intended to allow uninitialized state in constant evaluation, but left behind two bullets in the requirements for a constexpr function that still require explicit initialization for variant members.	Strike [dcl.constexpr]/4.1 and /4.2: — if the class is a union having variant members (11.4), exactly one of them shall be initialized; — if the class is a union-like class, but is not a union, for each of its anonymous union members having variant members, exactly one of them shall	Accepted

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
						be initialized; (This is CWG2424)	

US 055		09.02.3.5
		[dcl.fct]
			18	te	Defaulting an argument with a placeholder-type-specifier in an abbreviated function template is valid but useless. Calling the function without providing values for the default parameters will result in a deduction error.
					
					void foo(std::integral auto i = 0) {}
					
					foo(); // COMPILE FAILURE
					foo(1); // Ok
					
					This is also a problem for generic lambdas:
					
					auto foo = [] (auto i = 0) {};
					
					foo(); // COMPILE FAILURE
					foo(1); // Ok
						This problem could potentially be resolved if the invented template parameter was assigned an appropriate default value, e.g. decltype() of the provided default argument, although this could not work if the default argument references other parameters in the function.	Rejected
							There was no consensus to adopt this change.

US 056		09.03 [dcl.init]	n/a	te	C++20 intends to support parenthesized initialization of aggregates (eg, StructWithThreeElements(1, 2, 3)). The support for this also permits dropping trailing elements (eg, StructWithThreeElements(1)). Function-style casts from a single element are by definition equivalent to C-style casts, so this also unintentionally and problematically permits (StructWithThreeElements)1 and static_cast<StructWithThreeElements>(1). In addition, while the problem this feature is trying to address is real (container emplace is not sufficiently general), this change only tackles a corner of the problem (only aggregates), and in so doing, further increases the complexity of C++ initialization. This problem should be addressed by a big-picture consideration of C++ initialization; something like lazy parameters are	Revert the application of P0960R3. Even without the more general concerns about this feature, we do not have any confidence that a different change to initialization, designed at the last minute, will do any better than P0960 did, so we should not attempt to repair it, and should instead reconsider ways to tackle the problem for C++23.	Rejected There was no consensus to adopt this change.

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
					probably the right way to provide clean simple syntax for a generalized emplace.		
JP4 057		09.03.4	p3.10	ed	"const B &b2{a};" in the example is in inconsistent style on the place of &.	const B& b2{a};	Accepted - Editorial
US 058		09.04.1		ge	It is surprising (if consistent with if constexpr) that the bodies of non-templated functions whose constraints are not satisfied must be valid.	Make such function bodies discarded statements; change [stmt.if]/2 to treat discarded statements as non-instantiated templated entities that are not subject to [temp.res]/8.	Accepted with Modification See P1971
BG5 059	P 204-206	09.04.4		ge	There is a missed opportunity for consistency in naming get_return_object, initial_suspend and final_suspend. These 3 functions are very similar in the way, their return value is actually the customization point (and not so much the implementation of the functions themselves).	Rename initial_suspend to get_initial_suspend and final_suspend to get_final_suspend. Note: Is not just about consistency. initial_suspend and final_suspend can be very confusing to newcomers as they are novel customization concepts, that are worded similarly to regular observer functions. By having a get_ prefix - which observers in the Standard Library do not use - people will be reminded of get_return_object and the need to return a user-provided object as a customization.	Rejected There was no consensus to adopt this change.
BG6 060	P 204-206	09.04.4		ge	Most of the Promise function names are worded as observer functions, yet none of them are! Example: unhandled_exception reads exactly the same as uncaught_exception(s), already in the standard library! This will create confusion in people learning the API, as the same naming style is reused with a different meaning.	Mark functions, which are user implementations of required behavior, with an "on_" prefix. unhanded_exception becomes on_unhanded_exception, return_value becomes, on_return_value, return_void becomes on_return_void, yield_value becomes on_yield_value. This naming might seem novel, but the API itself is quite different from the rest of the Standard Library already. The fact that, for example, yield_value does not actually return the yielded value, the way std::begin does, is already a stark enough contrast that calls for different naming. Note: await_transform is a special, optional case and should be let as it is, unless the Committee decides otherwise.	Rejected There was no consensus to adopt this change.
US 061		09.04.4	10	te	Coroutine allocation does not consider std::align_val_t overloads introduced in C++17	Add them to the sequence of operator new calls that are attempted using wording similar to 7.6.2.7/18	Rejected There was no consensus

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
							to adopt this change.
US 062		09.04.4 [dcl.fct.def.coroutine]	5	te	Coroutine promise types are currently required to define an unhandled_exception member function. For many simple promise types, the definition of this function is simple and trivial; it just rethrows. It is unfortunate that authors of promise types are required to write this boilerplate. Many other coroutine extension points are only used if they exist, but this one is required. It would be nice if it was not required for noexcept coroutines.	If the coroutine is noexcept, do not wrap the invocation of the function body in a try/catch block and do not require that promise types define an unhandled_exception method.	Rejected There was no consensus to adopt this change.
US 063		09.04.4	9	te	The construction of the argument list for the call to the allocation function to allocate the 'coroutine state' does not call the overload of operator new() that accepts a std::align_val_t in the case that the allocation required for the coroutine has 'new-extended alignment'. This means that allocations of coroutine frames may not be correctly aligned in cases where the coroutine state contains overaligned types.	Apply similar wording from [expr.new]p18: Insert "If the coroutine state has new-extended alignment then the next argument is std::align_val_t." after "has type size_t." Insert at end of paragraph: If no matching function is found and the allocated coroutine state has new-extended alignment, the alignment argument is removed from the argument list, and overload resolution is performed again.	Rejected There was no consensus to adopt this change.
US 064		09.04.4 [dcl.fct.def.coroutine].	11	TE	Apply remaining coroutine TS issues to the working paper.	Apply resolution of the issue 33. “Parameter copy wording does not capture the intent.” From p0664r8	Accepted See p0664 issue 33
US 065		09.04.4 [dcl.fct.def.coroutine]	3	TE	Apply remaining coroutine TS issues to the working paper.	Apply resolution of the issue 24 “The specification of initial suspend point does not correctly captures the intent.” From p0664r8	Accepted with Modification See P1971
FR 066		09.05.4		te	Defining a coroutine’s promise type forces people who do not care about exceptions to write more boilerplate than necessary.	std::coroutine_traits<....>::promise_type::unhandled_exception should be optional and equivalent to exception re-throw when not declared.	Rejected There was no consensus to adopt this change.

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
FR 067		09.05.4		te	The number of customization points used by coroutines can be reduced while reducing boilerplate, increasing safety and consistency.	Adopt P1477 and remove coroutine_handle::from_promise (which is not needed after adopting P1477)	Rejected There was no consensus to adopt this change.
FR 068		09.05.4		te	promise_type::final_suspend can be renamed to promise_type::done to be more consistent with the proposed executor design.	Rename the customization point promise_type::final_suspend to done	Rejected There was no consensus to adopt this change.
US 069		09.06		te	Unnamed unscoped enumerations may be defined in multiple header units, and have no linkage. The merging rules of multiple definitions from header units or appearing textually outside of module purview require implementations to determine if two particular definitions are for the same entity. There is no mechanism specified to determine whether two such enumerations are for the same entity. Unnamed, *untypedefed*, enums are common in header files, as the enumeration values also appear in the containing (namespace) scope. A mechanism should be specified.	Use the first enumerator as the key. If two unnamed unscoped enumeration definitions in the same scope have the same identifier for their first enumerator, they are defining the same enumerated type. (It therefore is an ODR violation if the enumerators are not the same.)FYI this is the heuristic independently implemented in the Clang Modules extension and GCC C++ Modules. It is expected in Clang C++ Modules.	Accepted with Modification See P2115
US 070		09.06.2 [enum. udecl]		te	Keep the “using enum” language feature in C++20. This is a very useful feature.	If it helps consensus, consider the alternative spelling “using enumerators E;” instead.	Accepted
US 071		09.09.4 [dcl.fct.def.coroutine]	1	te	Coroutine keywords are prefixed with `co_` which looks ill-designed. Papers to rename them weren't seriously considered.	Adopt P1485R1.	Rejected There was no consensus to adopt this change.
US 072		1 - 15 Language		ge	Please address open CWG 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.	Accepted
US 073		10 and others		te	Because of the number of serious bugs that have been discovered in modules, including some that are requiring design changes at this late stage, modules do not seem sufficiently ready to be included in C++20.	Remove modules from the working draft with the intention to add them back soon after C++20 ships.	Rejected There was no consensus to adopt this change.

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
US 074		10 [module.unit]		te	Module names can contain ., but they have no semantic meaning. This seems to be a holdover from when they did have semantic meaning in previous designs. Keeping them will cause user confusion and may prevent future addition of semantics.	Disallow . in module-names. See P1873r0 for details.	Rejected There was no consensus to adopt this change. .
FR 075		10.01		te	As specified, modules names can contain spaces and comments module foo /* */.bar;	Specify that modules names are formed of a single entity with different rules than other identifiers. Consider not forbidding modules names containing C++ keywords between dots - but keep the ability to separate by dots so that modules can be organized logically.	Rejected There was no consensus to adopt this change.
US 076		10.01 [module. unit]		te	Remove *module-name-qualifier*. It removes our ability to have proper submodules, if we choose to do so at a later date.	Remove *module-name-qualifier*.	Rejected There was no consensus to adopt this change.
US 077		10.01		te	Allowing semantic less . in module names closes off the possibility of providing semantics in the future. Additionally . is a poor choice of separator in C++, where we currently use :: to represent the semantic less hierarchy used for code organization.	Disallow . in module names by removing the module-name-qualifier from the module-name grammar. Alternatively, replace the . in module-name-qualifier with ::	Rejected There was no consensus to adopt this change.
GB 078		10.01		Ed	Ensure consistent use of "digits" 'std + digits' being reserved is referenced in 3 places (wrt. modules and namespaces): [namespace.future] 16.5.4.2.3, [diff.cpp14.library] C.4.7, [module.unit] 10.1. In [module] "digits" appears in italics (indicating a definition by [intro.defs], though italic "digits" is also used in [cpp.include] 15.2). In [namespace] and [diff] "digits" does not appear in italics. Other references in the standard qualify "digits" with a base (e.g. "decimal digits"), or use it in a radix-neutral context (e.g. "number of digits"). Presumably the intention is that "digit" refers to the grammar in [lex.name] 5.10, e.g. that "stda" is	Either of: 1) Remove italic formatting from "digits" in [module.unit] 10.1; or 2) Apply italic formatting to "digits" in [namespace.future] 16.5.4.2.3 and [diff.cpp14.library] C.4.7 and optionally qualify "digits" with "decimal" (or other base as appropriate).	Accepted - Editorial

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
					not intended to be a reserved module name or namespace name and "std8" is. These references should be consistent.		
GB 079		10.01		Ed	The private-module-fragment is mentioned many times in the standard but its usage its obscure; as is in the papers. Nobody in the BSI knows its intended usage. Please provide an example.	Example added	Accepted See P1971
PL 080		10.01 [module.unit]		te	Dots in module names suggest a hierarchy, and that introduces a possibility of confusion among users. Several members of the committee have already suggested they intend to implement such a perceived hierarchy in the modules they ship, reexporting all "submodules" from the "main" module, and this will make it hard to invent other semantics for "submodules" in the future, and will also cause users to think that those semantics are a part of the language. We should allow ourselves more time to consider the possible semantics for actual hierarchical modules and submodules. Allowing dots in the module names makes it impossible for us to introduce such semantics. If we decide that we do not want to introduce hierarchical module semantics in the future, we can always relax the rules in the standard and allow dots in module names in the future.	Adopt P1873.	Rejected There was no consensus to adopt this change.
US 081		10.02	3	ge	Lifting the restriction on enclosing sequences of existing declarations that might include (say) a static_assert or might be rendered empty by the preprocessor, with "export {...}" in C++23, as suggested by SG2, would not help after existing code has been updated to use modules.	Apply the change recommended by §3 of P1766R1.	Rejected There was no consensus to adopt this change.
US 082		10.03 [module. import]		te	The order of initialization of globals in interface and header units is not defined for imported modules. This has the potential to break code, including iostream, when moving to modules.	Define an ordering. See P1874r0 for details.	Accepted See P1874
US		10.03		te	Header units make macros available to their importers. A header unit, A, may export an	Disallow exportation of header unit imports from named modules. Importers of a named module	Rejected

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
083					imported header unit, B – indeed, they cannot not export. An importer of A will observe B's exported macro set. However, if a named module, C, reexport imported header unit, B, C's importers will not observe B's macros via that path. This is confusing, and may make B's API unusable, without indirect importers directly importing B. This partial indirect importation has similar issues to today's unplanned reliance of indirect include files.	that relies on a header unit in its user interface will need to explicitly import that header unit.	There was no consensus to adopt this change.
US 084		10.03		te	export import foo; should not be allowed outside of module interface purview.	Ban using similar wording as 10.2/1	Accepted See P2109
US 085		10.03	6	ed	The paragraph discusses import declarations importing Translation Units. Translation Units are not imported, modules are (para 3). Para 5, describing header units, gets close to an import of a translation unit. The paragraph is describing the transitive nature of exported import declarations. The wording is confusing.	Reword paragraph 6 to avoid equating 'module' and 'translation unit'.	Rejected There was no consensus to adopt this change.
US 086		10.03 10.6	6 1	te	When a translation unit U imports translation unit T, the wording in 10.3/6 treats non-exported imports in T differently depending on whether T and U are in the same module. If they are in the same module those imports are also imported by U. This inconsistency is presumably to allow U to implicitly import implementation partitions imported by T, which cannot be exported. However this also applies to unrelated imports in T, including those within the global module fragment caused by implicit translation of #includes. This has an effect on the reachability rules in 10.6/1	A few options, some of which can be sensibly combined: 1. Apply this only to imports within the module purview of T 2. Apply this only for imports of partitions of the same module as T. Possibly only implementation partitions, because interface partitions could be directly exported 3. Eliminate this implicit import rule, and only implicitly import explicitly exported imports. 4. Eliminate the distinction between interface and implementation partitions. Allow any partition unit to be exported. Only require partition units containing export declarations to be exported by the primary interface unit.	Accepted with Modification See P1979

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
US 087		10.03	9	te	The wording restricts interface dependencies to module units. This allows header units to have cyclic dependencies and even allows them to be used to break dependencies among module units.	Replace all usages of “module unit” in 10.3/9 with “translation unit”	Accepted See P1971
US 088		10.04 [module. global], 15.4 [cpp.glob. frag]		ed	[module.global] and [cpp.glob.frag] refer to the same thing, but use different names.	Use the same name. Either glob.frag or global.	Accepted - Editorial
GB 089		10.06		Ed	Mark translation units in [module.reach] example [module.reach] 10.6 p5 The example in this paragraph comprises two translation units. They are currently separated by a blank line. These should be marked up as being separate translation units as per other examples, such as that in p4.	Add plain text line "Translation unit #1:" before "export module A;" Add plain text line "Translation unit #2:" before "module B;"	Accepted - Editorial
US 090		11.03.1	1	te	If P1815 is applied to C++20 (as is expected), but P1779 is applied to C++23 (as suggested by SG2), the latter will be a silent performance regression from changing language versions rather than being a concern to be addressed while converting header files to modules.	Apply P1779 (after appropriate CWG review).	Accepted See P1779
US 091		11.10.01	04.1	Te	A user-defined operator<=> can affect whether an enumeration type has strong structural equality.	Specify that all enumeration types have that property.	Rejected There was no consensus to adopt this change. .
US 092		11.10.01	04.2.1 [class. compare. default]	te	Class types with a non-static data member of array type do not have strong structural equality.	Require only that the (ultimate) element type of such an array type have strong structural equality.	Accepted See P1907
US 093		11.10.01	4	te	The definition of strong structural equality is used only in [temp.param].	Move it to [temp.param]	Accepted See P1907

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
US 094		11.10.01	4	te	Class types can have strong structural equality even if their operator== is deleted (since overload resolution "succeeds" even if it finds a deleted function).	Say "that is defaulted in the definition of C and is not defined as deleted".	Rejected There was no consensus to adopt this change.
US 095		12.02	1	ge	A declaration redeclares a constrained function if its requires-clause is equivalent. No atomic constraint expression can be equivalent to any other unless it is accessed via a concept (even within a single translation unit); for functions with no template parameters, all requires-clauses are functionally equivalent to either "requires true" or "requires false". In either case, the program is ill-formed NDR because of constructs that are functionally equivalent but not equivalent.	Document these severe restrictions, change the definition of (functionally) equivalent for atomic constraints to rely on the ODR, and/or eagerly evaluate non-dependent (portions of) constraints.	Accepted with Modification See P1971
CA 096		12.02 [over.dcl]	Paragraph 1	te	Declaration matching ([over.dcl]) is based upon whether trailing *requires-clause*s are equivalent; however, *equivalent*, with respect to expressions ([temp.over.link]), is defined only for expressions involving template parameters.	Extend the definitions of equivalent and functionally equivalent to cover expressions subject to normalization in general (not just those involving template parameters). Further, make the determination of expression equivalence treat concept definitions as opaque by adding a condition that an expression that may be subject to constraint normalization is functionally equivalent only if each *qualified-concept-name* that may be expanded by normalization would be considered to name the same type if, instead of a concept, a class template was named.	Accepted with Modification See P1980
US 097		13	1	Te	It does not seem useful to allow a type-constraint of Concept<> (directly, rather than via pack expansion).	Make the template-argument-list non-optional.	Rejected There was no consensus to adopt this change.
US 098		13	6	Ge	It is surprising that the single syntax Concept<X> can be a type-constraint (which becomes Concept<T,X>, not the Concept<X><T> that would result from adding <T> as for the other kind of type-constraint) or a very different id-expression.	Add a syntactic disambiguator, perhaps in the trivial form of Concept<,X> for the type-constraint case.	Rejected There was no consensus to adopt this change.

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
US 099		13.01	01.6	Te	"When a non-type template-parameter of non-reference and non-class type is used as an initializer for a reference, a temporary is always used." follows from its prvalue status.	Strike the sentence.	Accepted - Editorial
US 100		13.01	04.1	Te	Reference types (of which there are no glvalues) seem to vacuously have strong structural equality, which would allow an rvalue reference (which is a literal type) as a template parameter.	Explicitly exclude all reference types from strong structural equality.	Accepted with Modification See P1907
US 101		13.01	04.1	Ge	It is surprising that "template<int&,char&> int t;" is allowed, but that "struct S {int &i; char &f; bool operator==(const S&)=default/*delete*/;}; template<S> int t;" is not.	Don't use == to define the equivalence of class-type non-type template arguments (see comment on [temp.type]/1.5).	Accepted
US 102		13.01	04.1	Ge	If it is decided not to use == to define equivalence of class-type non-type template arguments (just as it is not used for references and pointers to members), some of the uncertainty surrounding non-type template parameters of floating-point type will no longer pertain, whereas the (very real) availability of undesirable workarounds involving std::bit_cast<int>(-0.f) will persist.	Apply P1714R1 (as already approved by EWG and CWG).	Accepted with Modification See P1907
PL 103		13.01 [temp.param]		te	The current syntax for constrained type template parameters, especially after the recent change of the naming convention for the standard library, causes confusion about the difference between the following two templates (one takes a value parameter, the other takes a constrained type parameter): template<bool B> struct foo {}; template<std::boolean B> struct foo {}; This is also inconsistent with the requirement to use the keyword `auto` for variable and parameter declarations with deduced constrained type:	In [temp.param]/1, replace the definition of the production rule *type-parameter* with: *type-parameter:* *type-constraint*$_{opt}$ *type-parameter-key* ...$_{opt}$ *identifier*$_{opt}$ *type-constraintopt type-parameter-key identifier*$_{opt}$ *= type-id* *template-head type-parameter-key* ...$_{opt}$ *identifier*$_{opt}$ *template-head type-parameter-key identifier*$_{opt}$ *= id-expression*	Rejected There was no consensus to adopt this change.

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<std::boolean auto B> struct foo {}; The author of this comment believes that, regardless of what the naming convention for standard library concepts ends up being, this is going to be confusing in real life code, and introduce a place where the knowledge of whether a name designates a type or a concept is necessary to be able to tell the *kind* of a template parameter. Therefore, this comment proposes that the second definition above would have to be written as following template<std::boolean class B> struct foo {}; In the future, if we decide that the perceived possible confusion between the first two definitions in this comment is not actually a problem, this can be further relaxed to allow the current syntax, simiarly to how `Concept auto foo` is expected to be possible to relax in the future if the committee finds that to be desirable.	Throughout the rest of the draft, replace all uses of `ConceptName TypeParameterName` with `ConceptName class TypeParameterName` (or `ConceptName typename TypeParameterName`).	
CA 104		13.04 [temp.constr]		te	The interaction between constraints and substitution has been the subject of some confusion. Declaration matching and partial ordering may require substitution that is not otherwise required to determine satisfaction; however, the wording does not make this clear in an accessible manner.	Add a least a note, likely with examples, indicating that declaration matching and partial ordering may require substitution into constraints. Since these substitutions are not being performed as part of determining viability of candidates for overload resolution, the SFINAE process does not apply.	Accepted See P2103
US 105		13.04.1	2	te	Nothing prohibits forming a pointer to a non-overloaded non-template function whose constraints are not satisfied.	Extend [over.over] to perform trivial overload resolution even when a function is named without a target type, obviating the need for [dcl.fct.def.delete]/2.	Accepted with Modification See P1972
US 106		13.04.1.2 [temp.constr.atomic]		ed	Concepts use the term "atomic", which is already a term of art within the C++ standard, as evidenced by clause [atomics].	Use a term other than "atomic" for concepts.	Rejected There was no consensus to adopt this change.
CA 107		13.04.1.4 [temp.constr.atomic]	Paragraph 2	te	The rules in 13.6.6.1 that the subject paragraph defers to does not handle parameter mapping for type template parameters, template template parameters, and non-type template parameters where substitution has made the expression non-	For non-dependent (after substitution) members of the parameter mapping, consider types by type identity, and expressions by type and value. P1624 describes a treatment for dependent cases that defer to the declaration matching rules	Accepted See P2103

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
					dependent.	through alias template and variable template proxies.	
US 108		13.04.2	2	Ed	The possibility of type-constraints appearing in a parameter-type-list is omitted (until /3.3.3).	Mention it alongside "template-parameter-list".	Accepted - Editorial
US 109		13.04.2	3	Te	Only templates are described as having associated constraints, but [over.match.viable]/3 and (via [temp.constr.order]/3) [over.match.best]/2.6 need them for non-template functions.	Replace "template" with "declaration"; other declarations will simply always match one of the first two bullets.	Accepted See P1971
CA 110		13.04.2 [temp.constr.decl]	Paragraph 3	te	Overload resolution ([over.match.best]) asks us to prefer a more constrained non-template function using rules that order declarations based on their associated constraints ([temp.constr.order]), but “associated constraints” are defined for templates ([temp.constr.decl]) and not for functions.	Add the following as a new paragraph before the subject paragraph: The *associated constraints* of a non-template function is the normal form of the *constraint-expression* introduced by the trailing *requires-clause*, if any; otherwise, the function has no associated constraints.	Accepted with Modification See P1971
US 111		13.04.3	1	Ge	It is surprising that the very special "pseudo)" evaluation semantics of && and || are not extended to !, and in particular that !A || !B is not at all the same as !(A && B) in case of substitution failure or for subsumption.	Assuming there is a rationale for the omission, add it as a note along with an example illustrating the failure of ! to invert a substitution failure.	Accepted with Modification See P1971
CA 112		13.04.4 [temp.constr.order]		te	How template parameters from one template is to be matched against template parameters in another template when they appear in substituted parameter mappings is not clearly defined.	In 13.6.6.2 [temp.func.order], candidates that are specializations of function templates should be ordered based on their constraints only when the templates have the same name (including for *conversion-function-id*s), parameter-type-list, and template parameter lists.	Accepted with Modification See P2113
US 113		13.05	01.5	Te	The == operator is inappropriate for comparing non-type template arguments of enumeration type, since it may be overloaded for them.	Compare the values of the underlying type.	Rejected There was no consensus to adopt this change.
US 114		13.05	01.5	Te	The == operator is inappropriate for template arguments of a class type with a member of	Approach #1: Forbid class types with members of such types (to make operator== equivalent to	Accepted with Modification

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
					enumeration or pointer-to-member type (and for object pointers would be incompatible with plausible extensions to [temp.arg.nontype]/2; see CWG 2043).	template-argument equivalence). Approach #2: Apply a suitable revision of P1837R0 that reverts not only P0732R2 but also part of P1185R2. Approach #3: Define equivalence of class-type non-type template arguments directly in terms of the (template-argument) equivalence of their base class subobjects and non-static data members (which allow to be references, but not mutable or volatile). Remove the definition of strong structural equality; restore from C++17's [temp.param]/4 bullets 1, 2, 4, and 5, or else use "a literal non-class type C for which, given an glvalue...". Directly forbid non-type template parameters of union-like class types.	See P1907
US 115		13.06.4 [temp.friend]		te	Hidden friends that are non-templates currently cannot have a requires-clause, but this functionality is important and used throughout Ranges.	Change [temp.friend]/9 to refer only to those friend declarations that are not any kind of templated entity.	Accepted with Modification See P2103
US 116		13.06.6.1	6	Te	There is no specification for equivalence among constraint-expressions.	Presumably, define it in terms of /5's expression equivalence.	Rejected There was no consensus to adopt this change.
US 117		13.06.6.1	6	Te	Types and type-constraints are supposed to be compared by /5, but it handles only dependent expressions.	Generalize /5 to support type-ids (by recursive decomposition). Compare non-dependent types by identity; compare non-dependent type-constraints according to the rules in [temp.type].	Accepted with Modification See P2103
US 118		13.07	8	Te	No reasonable implementation needs the freedom extended by making uninstantiable templates ill-formed with no diagnostic required.	With the exception of the last (long) bullet, specify instead that it is unspecified whether the program is ill-formed (with a required diagnostic) when the conditions pertain.	Rejected There was no consensus to adopt this change.
US 119		13.09.2.4	8	Te	The check that "deduction succeeds for a given type" suggests that each P/A pair is considered	Directly define "at least as specialized" in terms of the overall deduction succeeding, as seems to be	Rejected There was no consensus

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
					separately, contradicting [temp.deduct.type]/2.	the implementation consensus.	to adopt this change.
US 120		14.04.1.2	2	Te	There is no specification for which template parameters in (the template arguments in) one parameter mapping are "the same template parameter" (from [temp.over.link]/5) as those in another parameter mapping.	Use the mappings obtained by the partial-ordering deduction (which is required to have succeeded), augmented by matching by position for template parameters of the same kind that were not deduced per [temp.deduct.partial]/12 (to support constrained function templates like std::make_unique).	Accepted with Modification See P2113
US 121		15		te	P1703 requested in a change such that import declarations of header units were lexed as preprocessing directives – those for named modules were not. Import declarations (of either kind) are now preprocessing directives that result in tokens passed through to the C++ parser proper. The rationale for covering all import declaration was to permit source scanners operating in a non-standard preprocessing mode to extract module dependencies. This goal is not achieved. To achieve this goal, module declarations too must be treated as preprocessing directives. Without that, such scanners will not be able, in general, to detect module unit creation, only consumption.	Either: a) revert the changes inspired by p1703, or b) extend the changes inspired by p1703 to module-declarations, module-private-partitions and the global-module-fragment introducer.	Accepted with Modification See P1857
US 122		15		te	P1703 resulted in the creation of a preprocessing directive that does not begin with '#'. This is likely to confuse users, as the restricted lexing requirements come without the mnemonic '#' marker. It will also complicate code formatting tools, such as editors.	Revert the changes inspired by p1703	Rejected There was no consensus to adopt this change. .
US 123		15		te	Source scanners that do not use the complete preprocessing algorithm employ heuristics to approximate that. They will fail in some cases, whatever the standard specifies. Users of such source scanners already have to constrain the format of pieces of code to permit the scanner to function. The p1703 approach enshrines a particular scanning heuristic, with its own particular set of failing cases. For instance, when	Revert the changes inspired by p1703	Rejected There was no consensus to adopt this change.

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
					the scanner reaches an explicit header unit import, it will need to read the exported macros of that header unit. Correctly doing this requires reading the header's Compiled Module Interface. This may be done in several ways, amongst which are: 1) reading it directly, or 2) determining the CMI's exported macro set by processing the header unit's source file, or 3) approximating the exported set by reading the header unit's source file and discarding non-directive text, within the current scan. #1 requires interleaving of source scanning and compilation. The motivation of source scanners was to not do that.#2 is essentially implementing a macro-only module system inside the source scanner, which is liable to be both complex and/or inaccurate. #3 is an approximation, as it will observe macro definitions and undefinitions that are not exported. It will also incorrectly determine the prevailing macro definition algorithm of 15.3 in certain circumstances. A proposal to specify prevailing macro more in keeping with traditional #include ordering (p1174) was rejected. Thus approach #3 has been deemed undesirable. Alternative approaches of scanners overestimating the set of imports, but permitting failures, have been described more than once at meeting.		
US 124		15		te	P1703 relaxes the context sensitivity of the import keyword. Surrounding braces are no longer relevant – only the formatting of the line beginning with the import token (and possible preceding 'export' token). Previous drafts of the standard recognized the import keyword only outside of any braces (other than extern "C" linkage blocks). This requirement was motivated by p0795, which pointed out that 'import' and 'module' were used in the user interfaces of significant software. However, the import declaration's C++ grammar is unchanged, and it must appear at the outermost scope. The scope-agnostic lexing of the preprocessing directive will result in a) confusing errors at the parser level,	Revert the changes inspired by p1703	Rejected There was no consensus to adopt this change. .

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
					and b) frustration that such uses of 'import' must be locally protected from the preprocessor.		
US 125		15		te	The control-line changes to make import declarations be a directive were insufficient to achieve the stated goals of that change	Apply changes from P1857	Accepted with Modification See P1857
GB 126		15		Te	P1703R1 should be part of the Modules Tooling Technical Report, not part of the standard P1703R1 removed important features from import declarations: the context-sensitivity no longer takes braces into account and instead matches all lines starting 'import' (breaking compatibility with existing code — a codesearch.isocpp.org search for "import" finds many cases that will be broken by the new rule), and - line continuations are now required when import declarations span multiple lines (making use of attributes on import declarations ugly and awkward). There is also evidence that the proposal does not fully solve the problem that it aims to solve, as it does not cover module declarations.	Extend the new rules to also cover module declarations, allow import declarations to span multiple lines without backslash line continuations, and consider whether the context-sensitivity can be improved so that it doesn't reject the cases found by code search. Alternatively, revert P1703R1 from the C++ standard draft and instead establish a direction to include the rules from P1703R1 in the modules tooling technical report (as guidance on how to write code that supports dependency extraction from the widest possible set of tools).	Accepted with Modification See P1857
US 127		15 [cpp]	¶1	te	The import contextual keyword's context is too broad and breaks real code such as import->doImport(); .	Add one addtional token of context. See P1857r0 for details.	Accepted with Modification See P1857
US 128		15.01		te	A control line of the form '[export] import ...', is intended to be passed through to the C++ parser, after lexing and header-unit macro importation. However, this is never specified. The closest we get is in 15.3 where, for header unit imports, we specify that the 'import' keyword is replaced by a special token.	Specify (in 15.1, a new subsection, or make 15.3 more general) that these tokens are passed through.	Accepted with Modification See P1857
US 129		15.01	p1	TE	While the __has_cpp_attribute feature was under development in WG21, WG14 added C++-compatible attribute support to C2x and that was not taken into consideration for this feature. WG14 is considering adopting the same functionality for C but are having difficulties with the identifier chosen by WG21.	Rename __has_cpp_attribute to __has_attribute or some other language-agnostic name, or alternatively, keep the name __has_cpp_attribute and introduce a second, language-agnostic name as a synonym.	Rejected There was no consensus to adopt this change.

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
					While WG14 could always pick a name like __has_c_attribute, such a solution is unsatisfying because users would have to write twice as much code. C and C++ do not need separate preprocessor conditional inclusion features for this functionality – a single feature will suffice. e.g., #if __has_cpp_attribute(something) #define SOMETHING [[something]] #elif __has_c_attribute(something) #define SOMETHING [[something]] #endif is exactly equivalent to the shorter: #if __has_attribute(something) #define SOMETHING [[something]] #endif in cases where the code in question is shared between C and C++ compilers.		
US 130		15.01 [cpp.cond]	19	Te	Producing a token that might be reparsed as a 'defined' operator during macro replacement has undefined behavior. Undefined behavior lexing the program has no place in a modern standard, and this should either be a diagnosable error, or (perhaps conditionally) supported behavior to become that 'defined' operator.	Make this either a diagnosable error, or (perhaps conditionally) supported behavior to become that 'defined' operator.	Rejected There was no consensus to adopt this change.
US 131		15.02 [cpp.include]	4	Te	It is undefined behavior for token replacement to produce an include directive that does not match either of the two well-defined forms in the grammar. Undefined behavior lexing the program has no place in a modern standard, and this should be a diagnosable error.	Make this a diagnosable error	Rejected There was no consensus to adopt this change. .
US 132		15.03		te	Header units may provide macro definitions and undefinitions to their importers. These are #define and #undefs (of imported macros) that are encountered ‘when preprocessing each translation unit’. There is ambiguity as to whether this includes: a) macros defined on the command line, b) macros defined by the implementation (including indirectly via command line option), c)	Add wording to explicitly exclude these macro definitions (as there can be no imported macros visible at the point they are defined, explicit undefinitions are irrelevant).	Accepted with Modification See P1971

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
					macros defined in forced headers. Mailing list discussion concluded that such macros should NOT be exported.		
US 133		15.03		te	It is not clear whether header-unit source code can contain internal-linkage entities. For example the iostream header can contain: static ios_base::Ioinit __ioinit; Is that permitted, or does it make the header file incompatible with being a header unit?	Preference for internal linkage entities to be an error. Thus library implementors will need an (implementation-defined) mechanism to know whether the header file is being textually included, or whether it is being processed to create a header-unit.	Accepted with Modification See P1815
US 134		15.03		te	It is not clear whether header-unit source code can contain definitions of external linkage entities. For example: int version () { return 5;} 1) Does that emit a definition of 'version' to an object file associated with the header-unit? 2) Is it ill-formed? 3) Does it emit 'version' in the object file of each importer. #3 will lead to multiple-definition linker errors. At least 2 implementors of module compilers had differing understandings of this. Users will need to know whether an object file is a possibility.	Have a slight preference for permitting emission of an object file when creating a header-unit. i.e. option #1. However, option #2 would also be acceptable. Option #3 does not seem a good choice.	Accepted with Modification See P1815
DE 135		15.03	paragraph 2	te	"import" is not a language keyword to allow for backward compatibility with existing pre-modules source code, where "import" might be used as an identifier. The current status does not achieve the desired goal; for example "import->module = ENV;" on a line is considered as an (ill-formed) module import and cannot be parsed as an expression-statement.	Revert P1703R1. Document syntax restrictions to aid tools in the form of recommendations, outside of the C++ language standard.	Rejected There was no consensus to adopt this change.
US 136		15.04		te	The optional 'export' keyword of a module declaration must come from source file inclusion, as it is part of the pp-balanced-token-sequence. It may come from macro expansion. The first restriction is clearly an error. The second leniency is probably a difficulty for scanners.	Change the pp-global-module-fragment reduction to: module ; pp-balanced-token-seq export-opt module	Accepted with Modification See P1857
US 137		15.04		te	The post-phase-4 token sequences for module declarations specify that the module keyword	Either: 1) add an example showing such empty expansion is valid or 2) add text restricting the use	Accepted with Modification

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
					must not come from macro expansion. (And if other comments are accepted, this will be true in more cases, and for the export keyword too.) There is no restriction of interspersing NULL macro expansions, which again conflicts with the needs of source scanners. For instance: module; #define empty empty export empty module empty foo; Is this acceptable?	of such use	See P1857
US 138		15.04		te	The global module fragment grammar defines a pp-balanced-token-sequence. It is unclear whether the tokens of an import control line passed through to the c++ parser are part of the balanced sequence. (And therefore naked import control lines cannot appear in the GMF, and the effect of any unbalanced token sequence it might contain extends beyond the control line.)	Clarify that: a) the tokens of an import control line are, and b) the tokens of other control lines are not (because they do not emit pp-tokens). Note: Implementations might emit tokens to pass a pragma directive through, but the effect is as-if that is a single internal token.	Accepted with Modification See P1857
US 139		15.04 & 6.5		te	When there is no Global Module Fragment, a module declaration's tokens may be the result of macro expansion. When a GMF is present [10.4], the module token of the module declaration must not be the product of macro expansion. There is no restriction on the module keyword introducing a private module fragment in either case. This is at best inconsistent, and believed to be an error in conveying design intent. It presents difficulty with source scanners, that must therefore perform complete preprocessing to detect the module declaration in the non-GMF case. Therefore there is nothing to gain by the restrictions placed on the GMF.However, scanners could gain advantage if the restriction was applied to all module declarations. There is no implementation difficulty with either approach when compiling as specified in the std, it is purely for processing source code in an extra-standard manner. Compilers may have implementation difficulty detecting erroneous macro expansion generation as currently specified, when being given already-preprocessed tokens, as they	Either: a) The export & module tokens of a module-declaration, private module-fragment & global module fragment introducers must all never be the product of macro expansion, or b) No restriction on producing any export & module tokens from macro expansion.	Accepted with Modification See P1857

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
					usually cannot tell whether tokens are the result of macro expansion in that case (i.e. -fpreprocessed). This is compiling source in a manner outside the standard, so arguably not a defect.		
US 140		15.04/1		te	The requirements here apply only to files that lexically start with module; which means they have no effect for files where on entry to phase 7, the first token sequence forms a module declaration. Among other things, this allows a module declaration to come from an #include or macro expansion.	Require that TUs that don't start with module; either start with a module declaration at the start of phase 4, or they shall not contain any module declaration in phase 7. This would also be addressed by the changes in P1857	Accepted with Modification See P1857
US 141		15.05 [cpp.replace]	11	Te	It is undefined behavior to have a preprocessing directive inside the parens of a macro invocation. Undefined behavior lexing the program has no place in a modern standard, and this should be a diagnosable error, or (perhaps conditionally) supported behavior to immediately apply that directive. For example, existing practice on many compilers is to allow an if-section, although at least one compiler is known to diagnose an error in this case.	Make this either a diagnosable error, or (perhaps conditionally) supported behavior to immediately apply that directive.	Rejected There was no consensus to adopt this change.
US 142		15.05.2 [cpp. stringize]	2	Te	It is undefined behavior for token pasting with # to produce anything that is not a valid string literal. Undefined behavior lexing the program has no place in a modern standard, and this should be a diagnosable error.	Make this a diagnosable error	Rejected There was no consensus to adopt this change.
US 143		15.05.3 [cpp.concat]	3	Te	It is undefined behavior for token pasting with ## to produce anything that is not a valid preprocessing token. Undefined behavior lexing the program has no place in a modern standard, and this should be a diagnosable error.	Make this a diagnosable error	Rejected There was no consensus to adopt this change.
US 144		15.06 [cpp.line]	3	Te	If a #line directive is given a digit sequence outside the range 1..2,147,483,647 the behavior is undefined. Undefined behavior lexing the program has no place in a modern standard, and this should be a diagnosable error.	Make this a diagnosable error	Rejected There was no consensus to adopt this change. .
US 145		15.06 [cpp.line]	5	Te	If, after macro replacement, a #line directive does not exactly match one of the two supported forms, the behavior is undefined. Undefined	Make this a diagnosable error	Rejected There was no consensus

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
					behavior lexing the program has no place in a modern standard, and this should be a diagnosable error.		to adopt this change.
GB 146		15.10		Te	Concepts are missing a feature test macro There is no feature test macro for the concepts language facility.	Add a suitable definition of __cpp_concepts to [tab:cpp.predefined.ft]	Accepted with Modification See P1902
GB 147		15.10		Te	Add a feature-test macro for consteval	Add __cpp_consteval to Table 17 [tab:cpp.predefined.ft].	Accepted with Modification See P1902
US 148		15.10 [cpp. predefined]	4	Te	If a user attempts to #undef or #define a macro named 'defined', the behavior is undefined. Undefined behavior lexing the program has no place in a modern standard, and this should be a diagnosable error.	Make this a diagnosable error	Rejected There was no consensus to adopt this change.
US 149		15.10 [cpp. predefined]	4	Te	If a user attempts to #undef or #define a predefined macro named in this clause, the behavior is undefined. Undefined behavior lexing the program has no place in a modern standard, and this should be a diagnosable error.	Make this a diagnosable error	Rejected There was no consensus to adopt this change.
US 150		15.10 [cpp. predefined]	Table 17	GE	Familiar template syntax for generic lambdas should have a feature test macro: it is a significant enough feature	Add __cpp_lambda_template_parameters	Accepted with Modification See P1902
US 151		16 - 32 Library		ge	Please 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.	Accepted
US 152		16.03.4 [defns. comparison]		Te	This definition should be updated to accommodate the new 3-way comparison operator (7.6.8 [expr.spaceship]) as well.	Update the definition	Accepted See LWG Issue 3395
US 153		16.04.1.3 [structure.requirements] (library)	5	ed	P0898 applied a Cpp98 prefix (which was editorially changed to Cpp17) to all named requirements, in order to avoid ambiguity with library-defined concepts that had the same names. The named requirements are frequently part of user-facing library documentation, so changing their spelling carries a substantial	Delete the Cpp17 prefix from all named requirements, and update [structure.requirements] (16.4.1.3) paragraph 5 to reflect that change.	Rejected There was no consensus to adopt this change.

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
					educational cost. But after the application of P1754, the standard consistently uses capital letters in the spelling of all named requirements, and consistently avoids capital letters in the names of library-defined concepts, so the prefix is no longer necessary for disambiguation. In short, the benefits of the Cpp17 prefix have evaporated, but the costs remain.		
US 154		16.04.1.3 [structure.requirements] and many others	n/a	ed	P0898R3 applied a "Cpp98" prefix (which was editorially changed to "Cpp17") to all named requirements, in order to avoid ambiguity with library-defined concepts that had the same names. The named requirements are frequently part of user-facing library documentation, so changing their spelling carries a substantial educational cost. But after the application of P1754R1, the standard consistently uses capital letters in the spelling of all named requirements, and consistently avoids capital letters in the names of library-defined concepts, so the prefix is no longer necessary for disambiguation. In short, the benefits of the "Cpp17" prefix have evaporated, but the costs remain.	Delete the "Cpp17" prefix from all named requirements, and update 16.4.1.3/p5 to reflect that change.	Rejected There was no consensus to adopt this change.
GB 155		16.04.1.4		Ed	Consider renaming the "Expects:" and "Ensures:" elements in Library wording The choice of Expects: and Ensures: for library preconditions and postconditions was done for consistency with the C++ Contracts feature. Since Contracts are not in C++20, and if they return there's no guarantee that "expects" and "ensures" will be used, we should consider reverting to more conventional terms such as "preconditions" and "postconditions".	Change "Expects:" to "Preconditions:" and "Ensures:" to "Postconditions:" everywhere.	Accepted - Editorial
US 156		16.04.2.2.6	2	te	The text reads "The type of a customization point object shall model semiregular." However, the type of a customization point objecct is very likely to be const, and const types do not model semiregular. We should instead be testing the cv-unqualified type.	Should read, "The type of a customization point object ignoring cv-qualifiers shall satisfy semiregular." See https://cplusplus.github.io/LWG/issue3285	Accepted See LWG Issue 3285
US		16.05.1.2 [headers]	4	Te	The header <cstddef> should be added to the set of importable C++ library headers. It contains	Add <cstddef> to the list of importable headers.	Rejected

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
157					important C++ features like std::byte and std::nullptr_t that are more than just C compatibility. Likewise, the C compatibility layer is essentially important vocabulary typedefs that the C++ standard library relies on and are not otherwise exported from importable C++ library header units.		There was no consensus to adopt this change.
US 158		16.05.1.3 [compliance]		te	<coroutine> is listed as a freestanding header, however, it includes <compare>, which is not freestanding. Please ensure that <coroutine> is a freestanding header.	One possible resolution would be to make <compare> a freestanding header by adopting P1855. If that is not possible, <coroutine> could be modified to remove the dependency on <compare>. We do not consider making <coroutine> non-freestanding an acceptable solution.	Accepted with Modification See P1855
US 159		16.05.1.3 [compliance]		te	Please ensure that <compare> is a freestanding header.	Adopt P1855.	Accepted with Modification See P1855
GB 160		16.05.1.3		Te	<compare> should be in freestanding implementations. The <compare> header is closely tied to a language feature, and should be defined even for freestanding implementations.	Add <compare> to tab:headers.cpp.fs in [compliance].	Accepted with Modification See P1855
PL 161		16.05.1.3 [compliance]		te	<compare> is currently not a freestanding header. This causes two problems: 1. It is impossible to use the spaceship operator functionality in a minimal freestanding implementation. 2. The <coroutine> header, which is freestanding, uses <compare>, which is not.	Adopt P1855.	Accepted with Modification See P1855
US 162		16.05.3.5 [allocator.requirements]	[tab:allocator.req.var]	Te	The default behavior for a.destroy is now to call destroy_at	Replace "default" entry with: destroy_at(c)	Accepted
US 163		16.05.3.5 [allocator.requirement	[tab:allocator.req.var]	Te	The default behavior for a.construct is now to call construct_at	Replace "default" entry with: construct_at(c, std::forward<Args>(args)...)	Accepted

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
		s]					
FR 164		16.05.4.9		te	While char8_t, char16_t and char32_t are assumed to encode utf-8, utf-16 and utf-32 code units respectively, the encoding of u8string, u16string and u32string objects is not specified.	Adopt P1880	Rejected There was no consensus to adopt this change.
DE 165		16.05.5.4		te	It is unclear whether friend functions declared in a class are intended to be found via argument-dependent lookup only (and not via regular unqualified lookup), or whether the implementation is permitted to add declarations of that function that would allow unqualified lookup to succeed. For an example, see 17.11.2.2.	Clarify in the vicinity of 16.5.5.4 that friend functions are found via argument-dependent lookup only, unless a synopsis (but not a detailed specification, 16.4.1.4) expressly shows a namespace-scope declaration of that function. For existing friend functions, move non-trivial definitions from the synopses to regular descriptive elements.	Accepted with Modification See P1965
GB 166		17.03.1		Te	The new library span does not have a feature test macro	Add a definition of __cpp_lib_span to [tab:support.ft]	Accepted with Modification See P1917
US 167		17.03.1 Applies to Table 36 Standard Library Feature Test Macros [tab:support.ft]	Table 36	te	We forgot another feature test macro.	Add a new entry to the table: __cpp_lib_nonmember_signed_size | 201907L | <iterator>	Accepted with Modification See P1902
DE 168		17.03.1	Table 36	te	No consistent policy is applied to feature-test macros involving "constexpr" annotations and related features in the standard library. Currently, we have __cpp_lib_array_constexpr (note naming deviation)	Apply a consistent policy to constexpr-related library features: Either provide a single feature-test macro and remove all others, or create separate ones for each feature.	Accepted with Modification See P1902

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
					__cpp_lib_constexpr __cpp_lib_constexpr_dynamic_alloc __cpp_lib_constexpr_invoke __cpp_lib_constexpr_string __cpp_lib_constexpr_swap_algorithms __cpp_lib_constexpr_vector Some approved papers adding "constexpr" instructed to increase the value of the generic __cpp_lib_constexpr macro, others introduced separate macros. Paper P0202R3 instructed to add __cpp_lib_constexpr_algorithms, but that was apparently never reflected in the C++ Working Draft. Paper P1424R1 resolved to use __cpp_lib_constexpr for all constexpr-related library features, but that was apparently incompletely implemented.		
DE 169		17.08.2	paragraph 3	te	The expectation of the note that a default argument expression involving current() causes a source_location to be constructed that refers to the site of a function call where that default argument is needed has no basis in normative text. In particular, 9.2.3.6 paragraph 5 seems to imply that the name "current" and its semantics are bound where it appears lexically in the function declaration.	Add normative text to express the desired semantics.	Rejected There was no consensus to adopt this change.
US 170		17.11 [cmp]		te	The strong_equality and weak_equality comparison categories don't make sense now that we split equality from ordering. It doesn't make sense to declare an operator<=> that returns one of these – they just add needless complexity.	Remove strong_equality and weak_equality. Simplify three_way_comparable{,_with} to only deal with the ordering categories.	Accepted with Modification See P1959
US		17.11.02.1	4	te	Substitutability is ill-defined because it circularly	Either clarify the definition of substitutability, or	Rejected

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
171					depends on “comparison-salient state” and it is itself used to determine the correct return time of comparisons. Comparisons define what is “comparison-salient”, and if f can distinguish between a and b, it must be examining state that the comparison did not consider salient.	eliminate the distinctions between strong and weak comparisons, which are the only place that definition is used.	There was no consensus to adopt this change.
US 172		17.11.02.1	4	Ge	The concept of "substitutability" is meaningless: beyond issues like failing to require that f is pure and referring to "public const members" of something that might not be a class type, the only plausible definition of "comparison-salient state" is "any member of any notional tuple whose comparison is equivalent to that of the type", in which case everything has the property tautologically.	Replace the paragraph: "For the purposes of this subclause, a type T is said to exhibit substitutability if, given two values of type T such that a == b is true, a and b represent the same abstract value (as defined by T).". Alternatively, remove the definition as well as std::weak_equality and std::weak_ordering (which tellingly are never used except to propagate their use in user classes) and consider renaming std::strong_equality to std::equality and std::strong_ordering to std::total_ordering.	Rejected There was no consensus to adopt this change.
CA 173		17.11.02.2 [cmp.weakeq]		te	With the separation of <=> and ==, weak_equality has lost its primary use (of being a potential return type of <=>). Currently weak_equality serves no useful purpose in the standard (i.e., nothing in std acts on it), and just causes confusion (what’s the difference between weak and strong, when should I use which?) The difference between the two is ill-defined (involving substitutability and “salient” properties, which are also vaguely defined). The best definition of equality for a type is the type’s own == operator. We should not try to sub-divide the concept of equality.	Remove weak_equality and all references to it. Rename strong_equality to just equality. (New wording probably requires a paper, forthcoming).	Accepted with Modification See P1959
GB 174		17.11.04		Te	It's confusing for equality_comparable[_with] and totally_ordered[_with] to be in a completely different clause to three_way_comparable[_with]. We recommend moving [cmp.concept] to the same location as the others.	Move [cmp.concept] to Clause 18 [concepts.compare] and rename the sub-clause as [concepts.threewaycomparable] and have it included in <concepts>.	Rejected There was no consensus to adopt this change.
GB 175		17.11.06		Te	Move [cmp.object] to [comparisons] While it's nice to have <compare>, fragmenting	Move [cmp.object] to be a sub-clause of [comparisons] and have it (additionally) included	Agreed - Editorial

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
					function objects makes library organisation difficult. It would be good to migrate this type to [comparisons], where it will be with types that are similar.	in <functional>.	
US 176		17.11.06 [cmp.object] 20.14.7 [comparisons] 20.14.8 [range.cmp]		Te	The library defines a consistent total order for pointers in three places, but demands that only two of them be consistent. The total order in [comparisons] should be required to be the same total order as the other two subclauses. Ideally, this wording on the total order could be consolidated into one place, possibly in the clause 16 library-wide wording, and cross-referenced from these three places, simplifying the wording.	Make a consistent definition in a single place, and have all three uses refer to it.	Accepted See P1961
JP5 177		17.11.07	p1.3	ed	"ISO/IEC/IEEE 60599" is a typo.	ISO/IEC/IEEE 60559	Accepted - Editorial
CA 178		17.11.07 [cmp.alg]		te	std::strong_order, weak_order, and partial_order have special cases for floating point, but are missing special casing for pointers (whereas compare_three_way and std::less have the special casing for pointers)	1. Change [cmp.alg] bullet 1.4 from "Otherwise, strong_ordering(E <=> F) if it is a well-formed expression." to "Otherwise, strong_ordering(compare_three_way()(E, F)) if it is a well-formed expression." 2. Change [cmp.alg] bullet 2.4 from "Otherwise, weak_ordering(E <=> F) if it is a well-formed expression." to "Otherwise, weak_ordering(compare_three_way()(E, F)) if it is a well-formed expression." 3. Change [cmp.alg] bullet 3.3 from "Otherwise, partial_ordering(E <=> F) if it is a well-formed expression." to "Otherwise, partial_ordering(compare_three_way()(E, F)) if it is a well-formed expression."	Accepted See LWG Issue 3324

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
FR 179		17.12.03.2		te	coroutine_handle::from_address and coroutine_handle::address limit future evolutions while providing limited benefits	Remove the functions coroutine_handle::from_address and coroutine_handle::address	Rejected There was no consensus to adopt this change.
BG4 180	P 524	17.12.05	1	ge	(Related to BG2) The code example uses the void-returning variant of await_suspend().	Change suspend_never::await_suspend() to return its argument and change suspend_always::await_suspend() to return nullptr.	Rejected There was no consensus to adopt this change.
US 181		17-32		Te	The spaceship operator<=> is typically not usable unless the library header <compare> is directly included by the user. Many standard library headers provide overloads for this operator. Worse, several standard classes have replaced their existing definition for comparison operators with a reliance on the spaceship operator, and existing code will break if the necessary header is not (transitively) included. In a manner similar to the mandated library headers transitively #include-ing <initializer_list> in C++11, these headers should mandate a transitive #include <compare>.	Add: #include <compare> to the header synopsis for each of the following headers: <array> <chrono> <coroutine> <deque> <forward_list> <filesystem> <iterator> <list> <map> <memory> <optional> <queue> <ranges> <regex> <set> <stack> <string> <string_view> <system_error> <thread> <tuple> <type_index> <unordered_map> <unordered_set> <utility> <variant> <vector>	Accepted See LWG Issue 3330

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
US 182		18 [concepts]		te	P1754 changed the naming convention for concepts in standard library from PascalCase to snake_case. Using snake_case for standard library concepts creates confusion for users about which standard library facilities are concepts and which are concrete types. For example: ● function (type) & invocable (concept). ● iterator (type) & range (concept). ● iterator (type) & input_iterator (concept). ● bool (type) & boolean (concept). Please consider better ways of disambiguating standard library concepts from types, functions, and other kinds of things.	Possible resolutions include: ● Place all standard library concepts into a nested namespace, such as std::concepts. ● Add a Hungarian-notation-style prefix or suffix to standard library concepts, e.g. c_* or *_c. ● Be stricter about requiring that concept names be adjectives not nouns (for example, range is a noun). ● Use a different casing style for standard library concepts.	Rejected There was no consensus to adopt this change.
GB 183		18		Te	Adopt P1716 We're currently in a partial state between the old std::relation and what's currently in the CD. We should adopt P1716 to move to complete the change.	See P1716	Accepted See P1716
GB 184		18		Te	`object` should be a concept is_object_v is used in multiple places around the content that ranges introduces; it feels like a fundamental core concept, and we should probably introduce this as a concept so as to not shoot ourselves in the foot.	Add to [concepts]: template<class T> concept object = is_object_v<T>; Respecify movable so that it subsumes object. Respecify incrementable_traits, *cond-value-type*, iterator_traits, empty_view, single_view, ref_view, filter_view, transform_view, take_while_view, drop_while_view, and *semiregular-box*, to require object<T> instead of is_object_v<T>.	Rejected There was no consensus to adopt this change.
US 185		18.02 [concepts. equality]		te	This section talks about “implicit expression variations” but it isn’t actually clear what any of this wording means or how it is intended to be used. See also: https://github.com/ericniebler/stl2/issues/536 and https://github.com/ericniebler/stl2/issues/537	Clarify the meaning of the wording and its intended use.	Accepted with Modification See P2102
GB 186		18.02		Te	Rename "equality preserving" The term "equality preserving" is often called	Consider changing occurrences of "equality preserving" and "equality-preserving" and	Rejected There was no consensus

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
					"deterministic" in other contexts. Is there a better name here, particularly considering the negative of "not required to be equality preserving".	"equality-preservation" to something better, possibly based on 'deterministic'. Affects the following (sub)subclauses: [concept.convertible] [concept.commonref] [concept.swappable] [concept.equalitycomparable] [concept.invocable] [concept.regularinvocable] [range.cmp] [iterator.synopsis] [iterator.concept.readable] [iterator.concept.writable] [iterator.concept.winc] [iterator.concept.output] [range.range]	to adopt this change. Two different vectors can have different capacities.

GB 187		18.02		Te	What does equality-preservation imply for user-defined concepts?
					Expressions declared in a requires-expression in this document are required to be equality-preserving, except for those annotated with the comment "not required to be equality-preserving."
					While the wording concerns itself with standard concepts, it does not say anything about user-defined concepts. Should it be considered standard practice for user-defined requirements to be equality-preserving unless otherwise specified too?
					Example:
					template<typename T>
					concept spaceship_example_dont_use_me =
					std::regular<T> and
					std::totally_ordered<T> and
					requires(std::remove_reference_t<T> const& x,
					std::remove_reference_t<T> const& y) {
					x <=> y; // is this required to be equality-preserving too?
					};
						Provide clarification for whether or not user-defined concepts are required to be equality-preserving unless otherwise specified.	Rejected
							There was no consensus to adopt this change.

GB 188		18.02		Te	Is there a possible issue with stating that replacing a constant lvalue with a non-constant		Rejected There was no consensus

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
					lvalue should work? What if this involves binding to a const reference, where a non-const lvalue would fail (eg. a deleted overload)?		to adopt this change.
US 189		18.02	1	Ge	The "equal" relation used to define equality preservation (and modification) is completely unspecified (except implicitly when std::equality_comparable must be modeled), even for scalar types. It is useful for application code to rely on different definitions (e.g., comparing pointers or through them) for different algorithm calls; see also comment on [concept.moveconstructible]/1.	Specify that the relation (or the abstract value) is implicitly chosen by the program for each use of the library and that the library produces results consistent with the definition (so long as the associated semantic requirements are satisfied). Specify the (strongest) notion of equality supported by each language and library type that models the appropriate concepts.	Rejected There was no consensus to adopt this change.
US 190		18.02	2	Ge	Defining "domain" in terms of a requirement denies using it as a property of a type (e.g., in [concept.equalitycomparable]/1.1).	Treat separately the set of values supported and the set of values used (which are just the input values to an algorithm and any values it computes).	Rejected There was no consensus to adopt this change.
GB 191		18.04.13	01.2	Te	Clean up definition of "equal" Should English phrases such as (1.1) "u is equal to u2" be replaced by a definition using assertions on == or strong equality. What does "equal" mean for types with no == or <=> defined? How does this relate to equality_comparable?	Consider stronger wording for the definition of "equal" (e.g. "representationally equal").	Rejected There was no consensus to adopt this change.
US 192		18.04.13	1	Ge	With, for example, non-empty std::unique_ptr objects for which "equal" is defined by operator==, the move_constructible semantic requirements are vacuous since there is no equal u2 to consult.	Replace "equal" in [concepts.equality]/1 with an equally abstract notion of "value" (that is defined by the program and propagated by the library).	Rejected There was no consensus to adopt this change.
US 193		18.04.7 [concepts. arithmetic]		te	C++20 lacks a concept for arithmetic types. This omission is surprising, as this is a fairly common use case. For example, suppose I wish to write a function that squares a number. Pre C++20, I might write:	Change [concepts.arithmetic] (18.4.7) as follows: template<class T> concept integral = is_integral_v<T>; template<class T> concept signed_integral = integral<T>	Rejected There was no consensus to adopt this change.

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 T>
					auto square(T x) {
					return x * x;
					}
					
					In C++20, it would seem natural to be able to write:
					
					auto square(std::arithmetic auto x) {
					return x * x;
					}
					
					However, such a standard library concept is missing! Instead, we must write the more verbose:
					
					template <typename T>
					requires std::is_arithmetic_v<T>
					auto square(T x) {
					return x * x;
					}
						&& is_signed_v<T>;
						template<class T>
						concept unsigned_integral = integral<T>
						&& !signed_integral<T>;
						template<class T>
						concept floating_point =
						is_floating_point_v<T>;
						template<class T>
						concept arithmetic = is_arithmetic_v<T>;	

GB 194		18.04.7		Te	Respecify integral and floating_point Types that model integral or floating_point also model regular, and should refine regular at a minimum. To avoid overload resolution ambiguity, we should reconsider the definition of both concepts, and introduce two additional concepts: scalar and arithmetic, which form the basis of integral and floating_point. It was noted that this approach now causes there to be three times as many template instantiations for all integral and floating-point types. Given that there are a relatively small and finite number of integral and floating-point types, the author is not particularly concerned with this cost, especially as we move into a world of modules. It is unclear to the author whether or not the	Minimal change: template<class T> concept scalar = is_scalar_v<T> && regular<T>; template<class T> concept arithmetic = is_arithmetic_v<T> && scalar<T> && totally_ordered<T>; template<class T> concept integral = is_integral_v<T> && arithmetic<T>; template<class T> concept floating_point = is_floating_point_v<T> && arithmetic<T>; The preferred change is the same as the above,	Rejected There was no consensus to adopt this change.

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
					proposed change can be made after C++20 ships.	but refines arithmetic slightly further, to account for spaceship. template<class T> concept arithmetic = is_arithmetic_v<T> && scalar<T> && totally_ordered<T> && three_way_comparable<T>;	
US 195		18.05.2		te	The boolean concept is over-complicated and fails to capture what it intends because doing so would require it to be recursive (i.e., b satisfies boolean iff the expression b && b also satisfies boolean, etc.). LEWG decided in Cologne that the boolean concept should be removed and all uses of it in the library be replaced with convertible_to<bool>.	Remove the boolean concept and replace all uses of it with convertible_to<bool>.	Accepted with Modification See P1964
US 196		18.05.2 [concept. boolean]		te	The boolean concept is super complicated. Even still, it is possible to have two types that separately model boolean that still can't be used together. A simpler formulation would be easier to understand.	Consider the formulation: template <typename T> concept boolean = integral<remove_cvref_t<T>>; This won't accept true_type/false_type but at least means you can write conditions without bool casts throughout.	Accepted with Modification See P1964 .
GB 197		18.05.2		Te	Remove concept `boolean`, replace that requirement with `convertible_to<bool>` The concept boolean simultaneously: - has an overly-complex specification - is costly to check - doesn't achieve what it was designed to do ericniebler/stl2 #389 expands on more of this problem. The best solution forward is to remove std::boolean and replace it with std::convertible_to<bool>.	Strike [concept.boolean]. Replace all occurrences of boolean in the CD with convertible_to<bool>. Known occurrences: [cmp.concept] [concept.equalitycomparable] [concept.totallyordered] [concept.predicate]	Accepted with Modification See P1964
US 198		18.05.2	1	Ge	It is meaningless to have two different values b1 and b2 in the definition of the concept.	Rename b1 to b; remove b2 and use b (again) instead.	Accepted with Modification See P1964

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
GB 199		18.05.3		Te	Should equality-preservation concern itself with volatile or data races? Note text: should this mention volatile as well? Or data races?	Add a mention of volatile and/or data races to the note	Rejected There was no consensus to adopt this change.
GB 200		18.05.3	7	Ed	Example (7.2) is incomplete and doesn't show all of the combinations of pairs or arguments: there are 6 possible pairs but only 4 are shown. b == d and a == c are missing. Similarly, the a = c examples are incomplete.	Account for the missing examples.	Rejected There was no consensus to adopt this change.
US 201		18.05.4		te	The totally_ordered_with<T, U> redundantly requires both common_reference_with<const remove_reference_t<T>&, const remove_reference_t<U>&> and equality_comparable_with<T, U> (which also has the common_reference_with requirement). The redundant requirement should be removed.	Change the definition of totally_ordered_with to: template<class T, class U> concept totally_ordered_with = totally_ordered<T> && totally_ordered<U> && equality_comparable_with<T, U> && totally_ordered< common_reference_t< const remove_reference_t<T>&, const remove_reference_t<U>&>> && requires(const remove_reference_t<T>& t, const remove_reference_t<U>& u) { [...as before...]	Accepted See LWG Issue 3329
GB 202		18.05.4		Te	Define `totally_ordered[_with]` in terms of //`partially-ordered-with`// This will simplify the definition of both concepts (particularly totally_ordered_with), and make them in-line with equality_comparable[_with]. Now that we've defined partially-ordered-with for [cmp.concept], we should consider utilising it in as many locations as possible.	template<class T> concept totally_ordered = equality_comparable<T> && partially-ordered-with<T, T>; template<class T, class U> concept totally_ordered_with = totally_ordered<T> && totally_ordered<U> && common_reference_with<const remove_reference_t<T>&, const remove_reference_t<U>&> && totally_ordered< common_reference_t< const remove_reference_t<T>&, const remove_reference_t<U>&>> && equality_comparable_with<T, U> &&	Accepted See LWG Issue 3331

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
						partially-ordered-with<T, U>;	

GB 203		18.05.4		Te	`common_reference_with` requirement in `totally_ordered_with` is redundant.
					This is already required by equality_comparable_with, so by reshuffling the requirements, we can simplify the definition of totally_ordered_with.
						Change totally_ordered_with to:
						template<class T, class U>
						concept totally_ordered_with =
						totally_ordered<T> && totally_ordered<U> &&
						equality_comparable_with<T, U> && // moved up
						totally_ordered<
						common_reference_t<
						const remove_reference_t<T>&,
						const remove_reference_t<U>&>> &&
						requires(const remove_reference_t<T>& t,
						const remove_reference_t<U>& u) {
						{ t < u } -> boolean;
						{ t > u } -> boolean;
						{ t <= u } -> boolean;
						{ t >= u } -> boolean;
						{ u < t } -> boolean;
						{ u > t } -> boolean;
						{ u <= t } -> boolean;
						{ u >= t } -> boolean;
						};
							Accepted
							See LWG Issue 3329

FR 204		18.06	1	te	The concepts semiregular and regular require default constructibility. While default constructibility can be convenient in some cases, it can also be very harmful when there is no obvious default value for a type. Providing a default constructor for those types is a well known source of hard to find bugs, where the initial and meaningless value can be used as if it were a real one. Type with meaningless default constructor are even worse than use of uninitialized data, because this use can be detected by tools, while the use of meaningless data cannot. Regular is a nice name for something that should be fairly common, and adding a default constructibility requirement for regular will lead to many user types being default-constructible with no good reasons.	Remove the *semiregular* concept Change the definition of *regular* to: template<class T> concept regular = copyable<T> && equality_comparable<T>; Adjust the text in some places, such as: 16.4.2.2.6: The type of a customization point object shall model default_constructible and default constructible An alternative would be to totally remove both *semiregular* and *regular* from the standard, since anyways these concepts are not used much.	Rejected There was no consensus to adopt this change.

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
					See also https://quuxplusone.github.io/blog/2018/05/10/regular-should-not-imply-default-constructible/ for a more detailed discussion on this subject		
GB 205		18.07.3		Te	The regular_invocable name is potentially misleading as being related to the regular concept. Suggest pure_invocable or similar to indicate that neither the function or the arguments are changed. Since this is only a semantic difference from invocable then a clear name would help If the “equality preserving” term were changed to deterministic (or similar) then deterministically_invocable might be suitable (and contrasts nicely in meaning with non_deterministically_invocable).	Possible alternatives to regular_invocable: pure_invocable consistent_invocable Or deterministically_invocable deterministic_invocable If regular_invocable is renamed to either of these latter two, consider also renaming invocable to non_deterministic(ally)_invocable.	Rejected There was no consensus to adopt this change.
GB 206		18.07.3		Te	What is the intention for regular_invocable? The definition of regular_invocable states that calling invoke is equality-preserving and that neither the function object, nor its arguments are modified. A chat with Casey Carter in Cologne about why the function object can't also be const-qualified revealed that the intention of regular_invocable is to refine invocable so that it's equality-preserving. The author is not convinced that the current wording is in sync with this hallway discussion (by one of its designers). Examples: auto eq1 = [](auto const x) { return x * x; }; auto eq2 = [](auto& x) { x *= x; }; auto eq3 = [&x]{ x *= x; }; The author's understanding is that all three of these lambdas are equality-preserving, but only decltype(eq1) models regular_invocable.	Please confirm that the definition of regular_invocable is correct. If it is correct, please consider requiring that the function object also be const-qualified (this will help to prevent changes to the function object).	Rejected There was no consensus to adopt this change.
US 207		19.05.2.5 [syserr.errcat.objects]		te	The lifetime of the objects returned by functions like std::system_category() is unclear. Because these objects are meant to be referred to by std::error_code values, issues over the lifetime of the error category objects exposes use of	Provide a convenient mechanism to establish the lifetime of all similar error category objects associated with the implementation in one shot (perhaps in the style of [iostream.objects.overview]). Encourage	Rejected There was no consensus to adopt this change.

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
					std::error_codes during program termination ([basic.start.term]) to undefined behavior.	implementations to allow references to the associated objects as-if their lifetime began during constant initialization before that of any object with a non-trivial destructor.	
DE 208		20.02.1 [utility.syn]		te	Comparing and converting numbers of different numeric types is, should be a trivial task, but it's not because of implicit conversion P0586 was voted in Cologne, it adds free functions for comparing different numeric types as if they where signed types.	Adopt P0586 as discussed in Cologne and commented on GitHub (https://github.com/cplusplus/papers/issues/259)	Accepted See P0586
US 209		20.04.2 [string.view]		te	string_view should be made to be constructible from any contiguous character range in the new Ranges world.	Adopt P1391	Accepted See P1394 See P1391
CA 210		20.05 [tuple], 20.7 [variant], 31 [atomics], Annex D [depr.*]		te	Deprecate some uses of volatile in the standard library.	Adopt P1831.	Accepted with Modification See P1831
US 211		20.05 [tuple], 20.7 [variant], 31 [atomics], Annex D[depr.*]		te	Deprecate the library uses of volatile which were voted for deprecation by LEWG.	Adopt P1831R0.	Accepted with Modification See P1831
US 212		20.07.3.1 [variant.ctor]		te	Resolve LWG 3228: surprising variant construction	Resolve LWG 3228	Accepted with Modification See P1957
US 213		20.10.08.2	17	te	uninitialized_construct_using_allocator should use construct_at instead of operator new	*Effects:* Equivalent to: return ~~::new(static_cast<void*>(p))~~ construct_at(p, ~~T(~~make_obj_using_allocator<T>(alloc, std::forward<Args>(args)...)));	Accepted See LWG Issue 3321
US 214		20.10.10	1	te	propagation traits for std::allocator are inconsistent: POCMA and POCS should never	Add to allocator class definition:	Rejected There was no consensus

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

					differ	using propagate_on_container_move_assignment = true_type;
						using propagate_on_container_swap = true_type;
						using is_always_equal = true_type;
							to adopt this change.

US 215		20.10.11 [specialized. algorithms]	6	TE	The 'voidify' change introduced in 'The One Ranges Proposal' damages const correctness.	Delete *voidify*, and change places using *voidify* back to static_cast<void*>.	Rejected There was no consensus to adopt this change.
US 216		20.11.08 [util.smartpr. atomic] 31 [atomics]		ed	Please move the section specifying the atomic<shared_ptr<T>> and atomic<weak_ptr<T>> specializations from [utilities] (Clause 20) to [atomics] (Clause 31) so that it is located within the same section as the rest of atomic<T>. If this text is not relocated, it is more likely that the atomic<shared_ptr<T>> specializations will be overlooked in future changes to atomic<T>. We have encountered this same issue in the past with the <numeric> algorithms, which previously lived in [numerics], and were frequently overlooked when updates were made to [algorithms]. Moving the section that the text is in is purely an editorial change and does NOT imply changing which header the specializations are in.	Move [util.smartptr.atomic] (20.11.8) from [utilities] (Clause 20) to right after [atomics.types.memop] (31.8.5) in [atomics] (Clause 31). E.g. Make [util.smartptr.atomic] 31.8.6. The stable tag should not be changed.	Accepted - Editorial
US 217		20.12.03 & 20.12.3.2	2	te	polymorphic_allocator::allocate_object and new_object should be [[nodiscard]]	Add [[nodiscard]] in front of the return type for allocate_object and new_object in class declaration and in member-function description for polymorphic_allocator template.	Accepted See LWG Issue 3312
JP6 218		20.12.03.2	p1	ed	It's better to use a C++ property than C standard library macro, SIZE_MAX.	Replace "SIZE_MAX" with "numeric_limits<size_t>::max()"	Accepted with Modification See LWG Issue 3310
JP7 219		20.12.03.2	p8.1	ed	It's better to use a C++ property than C standard library macro, SIZE_MAX.	Replace "SIZE_MAX" with "numeric_limits<size_t>::max()"	Accepted with Modification See LWG Issue 3310
US 220		20.14.08	2	Te	The implementation-defined total order should be the same as that used by [comparisons].	State so.	Accepted with Modification See P1961

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
US 221		20.14.08	3, 7	Ge	Requiring the conversions to be equality-preserving is meaningless absent a definition of equality for the pointer type (which serves to constrain the definition for the parameter types).	Define it: in this case, in terms of the implementation-defined total order.	Rejected There was no consensus to adopt this change.
FR 222		20.15.10		te	std::is_constant_evaluated is easily misused, since it will always be true in if constexpr conditions,	Make std::is_constant_evaluated a language feature by adopting P1938 (if consteval {})	Rejected There was no consensus to adopt this change.
GB 223		20.20.02		Te	What does "not a format string" mean? std::format throws when the relevant argument "is not a format string", but [format.string] doesn't clearly say when a given input is "not a format string". Is "{a}" a format string consisting of those verbatim characters (because it doesn't match the grammar for a replacement field) or is it not a format string?	Define how "errors" in a format string are treated.	Rejected There was no consensus to adopt this change.
GB 224		20.20.02		Te	Format string grammar is in terms of narrow characters only The BNF grammar for format strings is specified in terms of char literals like '{' but it's not clear what that means for wide character strings such as L"{}".	Clarify the (obvious) mapping from wide characters to terminals in the grammar, i.e. L'{' is equivalent to '{' etc. Consider using the same grammar style as the core language, instead of a modified BNF.	Rejected There was no consensus to adopt this change.
GB 225		20.20.02.2		Ed	std::format() alignment specifiers should be independent of text direction The align specifiers for formatting standard integer and string types are expressed in terms of "left" and "right". However, "left alignment" as currently defined in the format() specification might end up being right-aligned when the resulting string is displayed in a RTL or bidirectional locale. This ambiguity can be resolved by removing "left" and "right" and replacing with "start" and "end", without changing any existing implementation and without changing the intent of the feature.	In [tab:format.align]: Forces the field to be ~~left-aligned within~~ <ins>aligned to the start of</ins> the available space and Forces the field to be ~~right-aligned within~~ <ins>aligned to the end of</ins> the available space	Accepted See LWG Issue 3327

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

GB 226		20.20.02.2		Te	Make locale-dependent formats for std::format() congruent with default formatting
					The design of format() prefers "locale-independent" formatting options for performance reasons. It provides very limited support for locale-dependent formatting via the 'n' specifier.
					It's particularly problematic that the 'n' specifier for floating point numbers is specifically limited to the chars_format::general presentation. It would be very useful to have access to chars_format::scientific and chars_format::fixed formatting with locale-dependent presentation.
					Adding these features to std::format() at this stage would require significant wording changes that are too large to contain in a comment. However, one approach that could be taken in the future would be to make 'n' be an additional suffix that could be added to format specifiers, rather than being a lone format specifier. This would enable locale-dependent formatting of any of the conversions of any of the arithmetic types.
					In order to keep the design space open for making this change in a future version of the standard, it would be ideal for 'n' conversions to always be congruent with the default conversion. It provides an intuitive semantic: 'n' is the same as "no specifier", but with locale-dependent presentation.
					The integer and charT presentation types currently specify 'n' conversions that are congruent with the default conversion.
					The bool and floating-point presentation types have 'n' conversions that are not congruent with the default conversion.
					For C++20:
					- Remove the 'n' conversion for bool.
					auto s format("{:n}", 1);
					// Committee Draft: s contains "1"
					// Proposed: ill-formed format string
					Making the 'n' conversion for floating-point match
						In [tab:format.type.bool]: Remove n.
						In [tab:format.type.float]: Replace the 'Meaning' of the n specifier with:
						If precision is specified, equivalent to to_chars(first, last, value, chars_format::general, precision), where precision is the specified formatting precision; equivalent to to_chars(first, last, value) otherwise. The context's locale is used to insert the appropriate digit group and decimal radix separator characters.
							Accepted with Modification
							See P1892

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

					the default conversion, i.e. dependent on whether a precision is specified.
					auto s format("{:n} {:2n}", 12.345678, 12.345678);
					// Committee Draft: s contains "12,3456 12,34"
					// Proposed: s contains "12,345678 12,34"
					These changes are the minimum necessary to allow enhanced support for locale-dependent formatting in the standard library to be added in a backwards-compatible way in a future edition of C++.		

US 227		20.20.02.2 [format. string.std]	5 Table 59	te	We believe that the lack of a way to suppress the negative sign on numbers which are rounded up to zero by the specified precision is a defect which will affect most users of format string.	Add: "'z' Indicates that a sign should not be used for negative numbers that display as zero (after rounding to the formatting precision)." To table 59. The details of the change will be proposed in P1496R1 in the pre-Belfast mailing. A "D" version of this paper was discussed in Kona this year.	Rejected There was no consensus to adopt this change at this time. The feature will be resubmitted for the next revision of C++.
US 228		20.20.02.2 [format.strin g.std]	Paragraph 7, Paragraph 9	te	Units of width and precision are not specified which causes an ambiguity for strings in variable-length encodings.	Width and precision for strings should be computed based on fixed operating system dependent encodings. If the operating system is capable of displaying Unicode text in a terminal both ordinary and wide encodings are Unicode encodings such as UTF-8 and UTF-16, respectively. [Note: this is the case for Windows-based and many POSIX-based operating systems. -- end note] Otherwise encodings are implementation-defined. For the given encoding, display width of a string is the number of column positions needed to display the string in a terminal [Note: This is similar to the semantics of the POSIX wcswidth function with a fixed encoding. —]	Accepted with Modification See P1868
GB 229		20.20.03		Te	Formatting functions don't allow throwing on incorrect arguments std::format is only allowed to throw if fmt is not a format string, but the intention is it also throws for errors during formatting, e.g. there are fewer arguments than required by the format string.	Allow exceptions even when the format string is valid. Possibly state the _Effects:_ more precisely.	Rejected There was no consensus to adopt this change.
DE 230		21.03.2.1		te	Because string::reserve() can no longer shrink the capacity,	add string::reserve() at the end of §4.2	Rejected

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
					it should be added at the end of §4.2 as one function where a non-const member function can not invalidate referencess, pointers, and iterators, if it does not grow the capacity		There was no consensus to adopt this change.
DE 231		21.03.3.5 ([string.erasure]) 22.3.8.5 ([deque.erasure]) 22.3.9.7 ([forward.list.erasure]) 22.3.10.6 ([list.erasure]) 22.3.11.6 ([vector.erasure]) 22.4.4.5 ([map.erasure]) 22.4.5.4 ([multimap.erasure]) 22.4.6.3 ([set.erasure]) 22.4.7.3 ([multiset.erasure]) 22.5.4.5 ([unord.map.erasure]) 22.5.5.4 ([unord.multimap.erasure]) 22.5.6.3 ([unord.set.erasure]) 22.5.7.3		te	The free erase/_if functions were moved from LFTSv2 to the IS, but P1115 fell through the cracks. It would be awkward to ship a version of free erase()/erase_if() with a known API issue (returning void instead of the number of elements removed), and then fix it up in a source-incompatible way in C++23.	Adopt P1115.	Accepted with Modification See P1115

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
		([unord.multiset.erasure])					
US 232		21.04.2 [string.view]		Te	Make string_view constructible from contiguous ranges of character type. This is an important integration with the new ranges facility, and should not be deferred to a later standard. We believe having basic_string_view be properly constructible from a range should be viewed not as a new feature but as fixing a "missing constructor" defect resulting from the integration of ranges and therefore feel this is in scope for NB comments.	Apply p1391	Accepted See p1391
US 233		21.07.3 [views.span]		te	span's constructors should be harmonized with the new Ranges concepts of contiguous_range and contiguous_iterator, needs to be done now.	Adopt P1394	Accepted See P1394
GB 234		22		Te	Adopt P1115R0 for C++20 P1209R0 added erase and erase_if functions for the containers. P0646R1 changed the remove members of list and forward_list to return the number of removed elements. We failed to coordinate these changes, meaning the non-member erase functions discard the useful information now returned by forward_list::remove. P1115R0 proposed to fix this, but isn't in the CD.	Adopt P1115R0 as an obvious defect in the new erase and erase_if functions. Note this affects multiple locations in clause 22	Accepted with Modification See P1115
US 235		22.02.7 [unord.reg]	11 17 Table 70	Te	C++20 design fix: the use of Hash::transparent_key_equal to enable heterogeneous lookup for unordered associative containers deviates from prior art, does not address the incompatibility concerns raised in the original LEWG review, and adds more subtle and confusing corner cases and will likely surprise and confuse the user. For details on the problem, see https://isocpp.org/files/papers/P1690R1.html#design-minimize-confusion P1690R0 proposed a fix that was reviewed by	See: P1690r1 For details on the problem, see https://isocpp.org/files/papers/P1690R1.html#design-minimize-confusion	Accepted See P1690

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
					LEWG in Cologne, which was forwarded to LWG for C++20 (poll results below) with a suggestion to prioritize as it would be a break to do later; unfortunately LWG ran out of time before getting to it. SF F N A SA 5 6 7 0 0		
US 236		22.02.7 [unord.req] et. al.	n/a	te	The working paper has an implementation of heterogeneous lookup that differs substantially from existing practice. Once we ship the design currently in the CD, we will have a difficult time retrofitting the design in P1690. LEWG reviewed and approved P1690 for C++20 in Cologne, but limited LWG review time prevented this from being moved.	Merge P1690 into the working paper	Accepted See P1690
PL 237		22.02.7 [unord.req]		te	Heterogenous lookup for unordered containers requires hasher to provide the transparent_key_equal nested type that denotes the predicate. This design is inconsistent with the method used for the ordered containers and existing non-standard implementations, that checks for nested is_transparent type. Furthermore, it prevents the implementation of generic hashers to be combined with dedicated type predicate. Finally, it overrides std::equal_to equality predicate, even in a situation when it is explicitly provided by the user.	Adopt P1690R0.	Accepted with Modification See P1690
US 238		22.02.7 [unord.req] [N4810]	11, 17 Table 70.	te	C++20 design fix: the use of Hash::transparent_key_equal to enable heterogeneous lookup for unordered associative containers deviates from prior art, does not address the incompatibility concerns raised in the original LEWG review, and adds more subtle and confusing corner cases and will likely surprise and confuse the user. For details on the problem, see https://isocpp.org/files/papers/P1690R1.html#design-minimize-confusion P1690R0 proposed a fix that was reviewed by LEWG in Cologne, which was forwarded to LWG	For proposed wording, see p1690R1	Accepted See P1690

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
					for C++20 (poll results below) with a suggestion to prioritize as it would be a break to do later; unfortunately LWG ran out of time before getting to it. SF F N A SA 5 6 7 0 0		
US 239		22.03.7.1	2	Te	There is no specification of whether std::array has strong structural equality.	Specify that it has no non-static data members other than the obvious array (and see also comment on [class.compare.default]/4.2.1).	Rejected There was no consensus to adopt this change.
FR 240		22.07		te	span::index_type's name is inconsistent with the convention used by other containers and views, notably string_view	Rename span::index_type to span::size_type	Accepted with Modification See P1872
US 241		22.07		te	Rename std::dynamic_extent to std::dyn, as repeatedly using such a long name in the upcoming mdspan proposal (P0009, slated for Library Fundamentals V3) is unnecessarily unwieldly.	Replace dynamic_extent with dyn throughout the subsections of 22.7.	Rejected There was no consensus to adopt this change.
US 242		22.07 [views]		Ed	This early view type should be editorially consolidated into the new section for views in general, rather than lying in the containers clause.	Move into 24 [Ranges]	Rejected There was no consensus to adopt this change.
FR 243		22.07.2		te	Both std::extent and the proposed std::static_extent are type traits, while std::dynamic_extent is not, which is surprising and inconsistent	Rename std::dynamic_extent to std::dynamic_extent_tag	Rejected There was no consensus to adopt this change.
FR 244		22.07.2		te	std::as_bytes and std::as_writable_bytes encourage undefined behavior	Consider removing these functions.	Rejected There was no consensus to adopt this change.
US 245		22.07.3		te	P1227R2 changed the size and indexing operations in span from the signed type ptrdiff_t to the unsigned type size_t. The typedef should	Replace index_type with size_type as per P1872R0.	Accepted See P1872

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
					be changed from index_type to size_type to be consistent and interoperable with the rest of the standard library.		
US 246		22.07.3 [views.span]		Te	Span should be constructible from a contiguous forwarding range or iterators thereof, and not just "Container" types with a data() member function. As this has the possibility of affecting overload resolution and SFINAE, it may not be possible to repair std::span in a later standard.	Apply P1394	Accepted See P1394
PL 247		22.07.3 [views.span]		te	span<T> provides a const-qualified begin() method and cbegin() method that produces a different result if T is not const-qualifed: 1) begin() produces mutable iterator over T (as if T*) 2) cbegin() preduces const iterator over T (as if T const*) As consequence for the object s of type span<T>, the call to the std::cbegin(s)/std::ranges::cbegin(s) produces different result than s.cbegin().	Change span<T> members cbegin()/cend()/crbegin()/crend()/const_iterator to be equivalent to begin()/end()/rbegin()/rend()/iterator respectivelly.	Accepted See LWG Issue 3320
PL 248		22.07.3 [views.span]		te	std::span uses the name `index_type` instead of `size_type` for the return type of its `size` function. There is a historical reason for this; std::span used to have a signed return type of `size`. This typedef is also used as a type for "index" or "count" parameters, but since they are all unsigned at this point, it seems like an unwarranted inconsistency with the rest of the standard library.	Either: 1. Rename std::span::index_type to size_type. 2. Add an additional alias, size_type, aliasing index_type, to std::span.	Accepted with Modification See P1872
US 249		22.07.3.1		te	Remove const_pointer and const_reference from span, as they are unused.	~~using const_pointer = const element_type*;~~ using reference = element_type&; ~~using const_reference = const element_type&;~~	Rejected There was no consensus to adopt this change.

PL 250		22.07.3.2 [span.cons]		te	The resolution of the LWG3101 prevents accidental undefined behavior caused by size mismatch between the range and constructed span, e.g.:
					void processFixed(span<int, 5>);
					void processDynamic(span<int>);
						Add 'explicit(extent != dynamic_extent)' specifier to the following constructors in [span.cons]:
						
						constexpr span(pointer ptr, index_type count);
							Accepted with Modification
							See P1976

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

					std::vector<int> v;
					processFixed(v); // ILL-FORMED after 3103, UB if v.size() != 5 before
					processDynamic(v); // OK
					However, the resolution does not prevent similar problems in the situation when the (ptr, len) or (ptr, ptr) constructor is used:
					processFixed({v.data(), v.size()}); // WELL-FORMED, UB if v.size() != 5
					processFixed({v.data(), v.data() + v.size()}); // WELL-FORMED, UB if v.size() != 5
					Morover, currently, the code remains ill-formed even if explicit cast is performed by the user:
					processFixed(span<int, 5>(v)); // ILL-FORMED
					To resolve the issue, the construction of fixed-size span from dynamic-sized range should be explicit:
					processFixed(v); // ILL-FORMED
					processFixed({v.data(), v.size()}); // ILL-FORMED
					processFixed({v.data(), v.data() + v.size()}); // ILL-FORMED
					processFixed(span<int, 5>(v)); // WELL-FORMED
					processFixed(span<int, 5>{v.data(), v.size()}); // WELL-FORMED
					processFixed(span<int, 5>{v.data(), v.data() + v.size()}); // WELL-FORMED
					To summarize:
					Source | Destination | Constructor
					Fixed | Fixed | Implicit, ill-formed if size-mismatch
					Fixed | Dynamic | Implicit
					Dynamic | Dynamic | Implicit
					Dynamic | Fixed | Explicit
						constexpr span(pointer first, pointer last);
						
						In the specification of constructors:
						
						template<class Container> constexpr span(Container& cont);
						
						template<class Container> constexpr span(const Container& cont);
						
						* Add 'explicit(extent != dynamic_extent)' specifier.
						
						* Remove 'extent == dynamic_extent is true' ([span.cons]p 14.1) from Constrains element.
						
						* Add 'If extent is not equal to dynamic_extent, then size(cont) is equal to extent.' to Expects element.
						
						In the specification of constructor:
						
						template<class OtherElementType, size_t OtherExtent>
						
						constexpr span(const span<OtherElementType, OtherExtent>& s) noexcept;
						
						* Add 'explicit(extent != dynamic_extent && OtherExtent == dynamic_extent)' specifier.
						
						* Replace the 'Extent == dynamic_extent || Extent == OtherExtent is true' constrain with 'Extent == dynamic_extent || OtherExtent == dynamic_extent || Extent == OtherExtent is true'.
						
						* Add 'If extent is not equal to dynamic_extent,	

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
						then s.size() is equal to extent.' to Expects element.	

PL 251		22.07.3.2 [span.cons]		te	C++20 introduced both std::contiguous_range and std::continous_iterator concepts, that are the generalization of the pointer to continuous sequence of objects, and type erased view for such ranges in form of std::span.
					However, these two features are not integrated together, as consequence std::span cannot be directly constructed from ranges that models std::contiguous_range and std::sized_range, nor from the pair of std::continous_iterator:
					std::vector v{...};
					std::span s = v; // OK
					std::span s = v | std::take_view(10); // ILL-FORMED
					
					std::span s(std::to_address(v.begin()), 2); //OK
					std::span s(std::to_address(v.begin()), std::to_address(v.begin() + 2)); //OK
					std::span s(v.begin(), 2); // IMPLEMENTATION-DEFINED
					std::span s(v.begin(), .begin() + 2); // IMPLEMENTATION-DEFINED
						Adopt P1394R3.	Accepted
							See P1394

CA 252		22.07.3.7 [span.objectrep]		te	as_writable_bytes standardizes UB. In particular, pointer interconvertibility between an object and its object representation (in array form) is not established. We should not hide reinterpret_cast inside another std function. Also, as_writable_bytes and as_bytes should not be free functions unless we plan on applying them to other std types (e.g., vector). Free functions should be designed as function overload sets or as functions acting on a concept (i.e., all containers or all views, etc.—not necessarily a C++ Concept).	Preferred: Remove as_writable_bytes and move as_bytes to be a member function of span. Alternative: Rename as_writable_bytes to something including the word “reinterpret”, such as reinterpret_as_bytes, and make it a member function (along with as_bytes as a member function).	Rejected There was no consensus to adopt this change.
US		23		te	The adoption of P1207 (movability of single-pass	Revert P1207, restoring the copyability	Rejected

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

253		24
		25
					iterators) has left the working paper in an inconsistent state. Many places both in [range.adaptors] and [algorithms] assume copyability of input and output iterators, an assumption P1207 invalidated by permitting input (but not forward) and output iterators to be move-only.
					
					For instance, here are three issues in filter_view alone:
					
					In [range.filter.iterator]/p5, the current_ member of a filter_view::iterator is copied:
					constexpr iterator_t<V> base() const;
					Effects: Equivalent to: return current_;
					
					In [range.filter.iterator]/p7 has the same problem:
					constexpr iterator_t<V> operator->() const
					requires has-arrow <iterator_t<V>>;
					Effects: Equivalent to: return current_;
					
					In [range.filter.iterator]/p8, we are copying out of the current_ member in a call to find_if:
					constexpr iterator& operator++();
					Effects: Equivalent to:
					current_ = ranges::find_if(++current_, ranges::end(parent_->base_), ref(*parent_->pred_));
					return *this;
					
					As an example from the [algorithms] clause, here is [alg.rotate]/p11, which is shown erroneously copying a potentially move-only output iterator:
					template<forward_range R, weakly_incrementable O>
					requires indirectly_copyable<iterator_t<R>, O>
					constexpr ranges::rotate_copy_result<safe_iterator_t<R>, O>
					ranges::rotate_copy(R&& r, iterator_t<R> middle, O result);
					Effects: Equivalent to:
					return ranges::rotate_copy(ranges::begin(r), middle, ranges::end(r), result);
						requirement to the weakly_incrementable concept.	There was no consensus to adopt this change.

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
					P1207 introduced an unknown but likely large number of bugs into the working paper. Tracking them all down would take time and leave us with little confidence that we had found them all. In addition, p1456 (Move-only views) was _not_ merged to the working draft, leaving us in the oddly inconsistent state where iterators could be move-only but views could not. This has caused yet more bugs. For instance, views::counted(first,n) returns a view that holds an iterator by value. If the iterator is move-only, then the resulting view is not a view because it fails to satisfy the copyability requirement of the view concept.		
GB 254		23		Te	Most of the ranges iterator operations should be marked [[nodiscard]] These are equality-preserving operations that return values always intended to be used. The library should reflect this.	Add [[nodiscard]] to the following operations. ranges::iter_move ranges::distance ranges::next ranges::prev	Rejected There was no consensus to adopt this change.
GB 255		23		Te	output_iterator and output_range shouldn't be concepts Very little uses these concepts, and it's not clear if they're actually necessary at all, since they're explicitly omitted in N3351 (see §3.7). The author's understanding is that they were added to mitigate potential confusion among users familiar with STL output iterators. We should be judicious about the concepts that we introduce. If an output_iterator concept proves itself to be useful, then we can probably add it in C++23.	Proposed Change: Strike [iterator.concept.output] and associated references. Strike output_range from [range.refinements] or transform it into an exposition-only output-iterator. Re-specify the following algorithms so that they require weakly_incrementable && writable instead of output_iterator, or to require *output-iterator* (an exposition-only concept): replace_copy replace_copy_if fill fill_n	Rejected There was no consensus to adopt this change.
GB 256		23		Te	iterator concepts belong in namespace ranges This will (hopefully) help solidify that the iterator concepts introduced by C++20 aren't a one-to-one mapping between STL iterators and ranges	Move all concepts in [iterators] into namespace ranges. Rename input_or_output_iterator to iterator.	Rejected There was no consensus to adopt this change.

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
					iterators (similarly to how ranges algorithms aren't identical to std algorithms). It'll also let us use the name iterator instead of input_or_output_iterator.		
US 257		23.02 and 24.2		te	Two of the ranges opt-in variable templates are negative and checked against, the other is positive and checked for. Double negatives are needlessly difficult to understand. Make all the opt-in variable templates enable_meow instead of disable_meow.	Adopt P1871.	Accepted See P1871
US 258		23.03.1 [iterator.requirements.general]	10	Ed	It is unhelpful for the library to overload the definition of *reachable* with the core language definition of reachable for modules. Based on usage throughout this clause, suggest including the following 'from' in the defined words of power. This is consistent with every intended use of the current term, and no subsequent usage requires (nor uses) italics on either word.	Change font to italics on the word 'from': is called *reachable* from an iterator i to is called *reachable from* an iterator i	Accepted - Editorial
US 259		23.03.2.3	03.3	te	Types satisfying input_iterator but not equality_comparable look like C++17 output iterators. This issue is discussed in detail in LWG#3283.	Adopt the proposed resolution at https://cplusplus.github.io/LWG/issue3283	Rejected There was no consensus to adopt this change at this time. LWG Issue 3283 has been opened for future consideration, post C++20.
US 260		23.03.2.3/p4 23.5.4.2/p1 24.7.4.3/p3 24.7.7.3/p3 24.7.8.3 24.7.8.5/p1		te	It is currently impossible to non-intrusively opt-out of conformance to the C++17 iterator concepts without also opting out of conformance to the C++20 iterator concepts. This is a corner case that was missed when the Ranges TS was merged into namespace std. The issue is discussed in depth in LWG#3289.	Adopt the proposed resolution in https://cplusplus.github.io/LWG/issue3289	Rejected There was no consensus to adopt this change at this time. LWG Issue 3289 has been opened for future consideration, post C++20.
US 261		23.03.4.13	02.6	te	The expression ++(a + D(n - 1)) is erroneously applying pre-increment to an rvalue iterator. This is not required to be valid for random access iterators.	Replace with either []([I c){ return ++c; }(a + D(n - 1)) or with next(a + D(n - 1)). See https://cplusplus.github.io/LWG/issue3277	Accepted with Modification See P1917

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
US 262		23.03.4.13	02.7	te	The semantic constraints of the random_access_iterator concept is accidentally promoting the difference type using unary negate.	Change (b += -n) to (b += D(-n)). See https://cplusplus.github.io/LWG/issue3284	Accepted See LWG Issue 3284
US 263		23.03.4.2		te	In the current spec, shared_ptr<int> is readable, but shared_ptr<int>& is not. That is because readable_traits is not stripping top-level references before testing for nested typedefs.	Change every occurance of iter_value_t<In> in the definition of the readable concept with iter_value_t<remove_reference_t<In>>. See https://cplusplus.github.io/LWG/issue3279	Accepted See P1878
US 264		23.03.4.2		te	The readable concept is both under- and over-constrained. It is under-constrained in that it permits its associated types (iter_value_t, iter_reference_t, etc) to differ depending on whether the type is const-qualified or not. It _might_ make sense for iter_reference_t to be sensitive to const-ness if, for example, it is our intention for a type like optional to satisfy readable. Generally we use readable to constrain types that are logical indirections; e.g., pointers (smart and dumb) and iterators. For those, top-level cv-qualification should not matter. readable is over-constrained because it only requires operator* to be valid on a (non-const) lvalue. See discussion at https://github.com/ericniebler/stl2/issues/514	Change the definition of the readable concept to correct these problems.	Accepted See P1878
FR 265		23.03.4.6		te	input_or_output_iterator does not denote input_iterator<It> || output_iterator<It>, which sets a bad precedent for concept naming and may not match the user intent. It is also at odds with the naming used for decades including most literature and Stepanov's work.	Rename it to general_iterator	Rejected There was no consensus to adopt this change.
US 266		23.03.5.3	Table 85	Te	What does it mean for an output iterator to be incrementable after any number of increments?	Add a note explaining the satisfaction of the property from [iterator.concept.winc]/13, provide an alternate definition, or remove the Ensures	Rejected There was no consensus to adopt this change.

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
US 267		23.03.6.2 indirectcallable. indirectinvocable, others		te	The ranges compare algorithms are over-constrained. LEWG approved P1716 with the correct fix, but LWG ran out of time to review it. Without P1716, safe and correct programs will be erroneously rejected.	Adopt the proposed resolution in P1716, which has already passed LEWG design review.	Accepted See P1716
US 268		23.03.7.4		te	The indirectly_swappable concept is over-constrained: it requires only that iter_swap is callable with lvalue iterators. It should be possible to call iter_swap with rvalues as well. See discussion at https://github.com/ericniebler/stl2/issues/578.	Change the concept to require iter_swap to be callable with both lvalue and rvalue iterators.	Accepted See P1878
DE 269		23.07 Range access	Paragraph: 18	te	There are ranges that model std::ranges::sized_range, but do not provide a .size() member function. (Also) for this reason std::ranges::size() was introduced with slightly different semantics than std::size(). Now we are introducing std::ssize() with the semantics of std::size() + signed-ness. This means we get three size functions that each have different deficiencies and none that works for all sized ranges and is signed.	Preferred: Make std::ssize() resolve to std::ranges::size() + signed-ness. Alternative: Also add std::ranges::ssize().	Accepted with Modification See P1970
GB 270		24		Te	P1207 provided an opportunity for us to weaken input iterators so that they don't need to be copyable. While the author thinks that this is a step in the correct direction, P1207 has left us in a partial state where iterators don't need to be copyable, but views do. Given that views have underlying iterators, we need to address this problem before C++20 ships.	Either apply P1456 and evaluate all standard range adaptors to determine if they're affected (and then apply changes to bring them into accordance with P1207 and P1456), or completely roll back P1207.	Accepted See P1862 See P1456
GB 271		24		Te	Many operations in namespace ranges should be marked nodiscard These are equality-preserving operations that return values always intended to be used. The library should reflect this intention in the strongest way possible.	Add [[nodiscard]] to the following operations. ranges::begin ranges::end ranges::cbegin ranges::cend ranges::rbegin	Rejected There was no consensus to adopt this change.

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
						ranges::rend ranges::crbegin ranges::crend ranges::size ranges::empty ranges::data ranges::cdata Add [[nodiscard]] to the following view_interface member functions. empty data size front back operator[] Add [[nodiscard]] to the following subrange member functions. begin end empty Size	
US 272		24 Applies to §24 Ranges [ranges]	§24.4.2 Ranges [range.range]; §24.6.1.2 Class template empty_view [range.empty.view]; §24.6.3 Class template iota_view[range.iota.view]; §24.7.6.4	te	Due to the ranges API being more or less fixed after shipment, API-breaking fixes have to be scheduled now rather than shipped in C++23. The papers p1664 and p1739 represent important and ultimately source-breaking changes to the ranges API. Unless shipped, these API optimizations will result in source code breaking at a later date if attempted to be fixed later, and also makes it impossible to reliably simplify the return value of ranges for a wide variety of current and future adaptors and algorithms. These changes are imperative for developing better APIs and it would be unfortunate to not be able to do them post-C++20 due to source breaking changes. All of the changes except for p1664's two new exposition-only concepts have been approved by LEWG during review of p1739. However, p1664 was discussed as part of p1739's approval and this comment was to be expected.	The fix and its motivations have been formalized in two papers, [p1664 - Reconstructible Ranges] and [p1739 - Type preservation for forwarding Ranges for "subrange-y" views]. Further fixes are applied through Corentin Jabot's and Casey Carter's [p1391] and [p1394].	Accepted with Modification See P1739

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
			views::take [range.take.adaptor]; §24.7.8.3 views::drop [range.drop.adaptor].				
US 273		24.02		Te	all_view is not a view like the others. For the other view types, foo_view{args...} is a valid way to construct an instance of type foo_view. However, all_view is just an alias to the type of view::all(arg), which could be one of several different types. all_view feels like the wrong name.	Suggest renaming all_view to all_t and moving it into the views:: namespace.	Accepted See LWG Issue 3335
GB 274		24.02		Te	Add range_size_t LEWG asked that range_size_t be removed from P1035, as they were doing a good job of being neutral w.r.t whether or not size-types were signed or unsigned at the time. Now that we've got a policy on what size-types are, and that P1522 and P1523 have been adopted, it makes sense for there to be a range_size_t.	Add to [ranges.syn]: template<range R> using range_difference_t = iter_difference_t<iterator_t<R>>; + template<sized_range R> + using range_size_t = decltype(ranges::size(declval<R&>()));	Accepted See P2091
GB 275		24.03		Te	ranges::begin and ranges::end should not accept arrays of unknown bound The current definitions of ranges::begin and ranges::end mean that an array of unknown bound is treated as an empty range. The expressions E+0 and E+extent_v<T> are both well-formed for an array of unknown bound (with extent_v<T> equal to zero).	Make ranges::begin(E) and ranges::end(E) ill-formed when E is an array of unknown bound.	Accepted with Modification See P2091

US 276		24.03.1
		24.3.2
		24.5.3
		24.7.3.1
		24.6.3.2
		several
			01.3
			1.3
				te	Several of the range views define non-template friend function begin/end overloads taking rvalues to satisfy the exposition-only forwarding-range concept. These have a couple of problems. First, the ones for subrange take subrange&&. That means that a const rvalue subrange fails to satisfy forwarding-range, which causes cbegin(subrange{...}) to be ill-formed.	In [range.access.begin]/p1.3, change the poison-pill overloads from:
						template<class T> void begin(T&&) = delete;
						template<class T> void begin(initializer_list<T>&&) = delete;
						
						...to:
							Accepted with Modification
							See P1870

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

					The bigger problem is that since these functions are non-templates, whenever they get added to the overload set, the compiler will try conversions to these types (subrange, ref_view). The attempted conversions could lead to errors in theory.
					
					Finally, class iota_view has iterator that can safely outlive the view that created them, so it too should be given begin/end friend functions that accept rvalues following the same pattern.
						template<class T> void begin(T&&) = delete;
						template<class T> void begin(initializer_list<T>) = delete;
						
						To the synopsis in [range.subrange]/p1, add:
						template<class A, class B>
						concept same-ish = // exposition only
						same_as<A const, B const>;
						
						In the class synopsis of subrange (same section), change the `begin`/`end` friend functions from:
						friend constexpr I begin(subrange&& r) { return r.begin(); }
						friend constexpr S end(subrange&& r) { return r.end(); }
						
						...to:
						friend constexpr I begin(same-ish<subrange> auto && r) { return r.begin(); }
						friend constexpr S end(same-ish<subrange> auto && r) { return r.end(); }
						
						In the synopsis of `ref_view` in [range.ref.view]/p1, change the `begin`/`end` friend functions from this:
						friend constexpr iterator_t<R> begin(ref_view r)
						{ return r.begin(); }
						friend constexpr sentinel_t<R> end(ref_view r)
						{ return r.end(); }
						
						...to this (editors note: the use of same_as here instead of same-ish is intentional; likewise for the use of pass-by-value):
						
						friend constexpr iterator_t<R> begin(same_as<ref_view> auto r)	

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
						{ return r.begin(); } friend constexpr sentinel_t<R> end(same_as<ref_view> auto r) { return r.end(); } To the class synopsis of iota_view in [range.iota.view], add the following `begin`/`end` friend functions: friend constexpr W begin(same-ish<iota_view> auto && r) { return r.begin(); } friend constexpr auto end(same-ish<iota_view> auto && r) { return r.end(); } See https://github.com/ericniebler/stl2/issues/592.	
GB 277		24.04		Te	Adopt P1456 or change istream_view's requirements P1456 weakens view so that it does not require copyable. Without this, istream_view is unable to process non-copyable types as input. LEWG approved P1456 in Kona, but it seems that it didn't make it in time for LWG in Cologne. I don't think this is something that can be fixed in C++23.	Apply the proposed wording in P1456 to the International Standard.	Accepted with Modification See P1456
GB 278		24.04		Te	Rename viewable_range to viewable The name viewable_range doesn't communicate its intended purpose very clearly. Consider renaming to viewable, which very clearly reflects its description, and thus intended purpose. You can't convert a non-range to a view anyway.	Rename viewable_range to viewable. Affects: [ranges.syn] [range.adaptor.object] [range.filter.view] [range.transform.view] [range.take.view] [range.join.view] [range.common.view] [range.reverse.view]	Rejected There was no consensus to acopt this change.
US 279		24.04.2 [range. range]		te	The *forwarding-range* concept opt-in is too subtle and just adds complexity to overload resolution. The other range concepts use variable templates to opt-in, this one should do.	Adopt P1870.	Accepted with Modification See P1870

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
GB 280		24.04.2		Ed	forwarding-range is too-easily confused with forward_range	Please find an alternative name for forwarding-range, as it is extremely similar to forward_range, which is (a) different in definition, and (b) a user-facing concept.	Accepted See P1870
FR 281		24.04.4		te	The View concept requires copy-ability. There are reasons why this is overly restrictive - for example a predicate might not be copyable or a view might need to hold a coroutine_handle which should not be copied. As concepts are hardly modifiable after the publication of the standard, it is important to relax this constraint while we still can	Adopt P1456 which was approved by LEWG	Accepted with Modification See P1456
DE 282		24.04.4		te	"Since the difference between range and view is largely semantic, the two are differentiated with the help of enable_view." (§3) enable_view is designed as on opt-in trait to specify that a type is a view. It defaults to true for types derived from view_base (§4.2) which is clearly a form of opt-in. But it also employs a heuristic assuming that anything with iterator == const_iterator is also view (§4.3). This is a very poor heuristic, the same paragraph already needs to define six exceptions from this rule for standard library types (§4.2). Experience in working with range-v3 has revealed multiple of our own library types as being affected from needing to opt-out from the "auto-opt-in", as well. This is counter-intuitive: something that was never designed to be a view shouldn't go through hoops so that it isn't treated as a view.	Make enable_view truly be opt-in by relying only on explicit specialisation or inheritance from view_base. This means removing 24.4.4 §4.2 - §4.4 and introducing new §4.2 "Otherwise, false". Double-check if existing standard library types like basic_string_view and span need to opt-in to being a view now.	Accepted See LWG Issue 3286
US 283		24.05.1		te	The exposition-only has-arrow concept is ill-formed. It has a constrained template parameter, which is not valid C++20.	Change the concept to: template<class I>	Accepted with Modification

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
						concept has-arrow = // exposition only input_iterator && (is_pointer_v<I> || requires(I i) { i.operator->(); });	See P1983

US 284		24.05.3		te	Conversion from pair-like types to subrange is a silent semantic promotion. Just because a pair is holding two iterators does not mean those two iterators denote a valid range. Permitting that pair to be implicitly converted to a subrange is error prone.	In the synopsis of subrange, strike the definition of the exposition-only _pair-like-convertible-to_ concept, and the following two subrange constructors:
						
						template<not-same-as<subrange> PairLike>
						requires pair-like-convertible-to<PairLike, I, S>
						constexpr subrange(PairLike&& r) requires (!StoreSize);
						
						template<pair-like-convertible-to<I, S> PairLike>
						constexpr subrange(PairLike&& r, make-unsigned-like-t(iter_difference_t<I>) n)
						requires (K == subrange_kind::sized);
						
						See https://cplusplus.github.io/LWG/issue3281
							Accepted
							See LGW Issue 3281

US 285		24.05.3	1	te	The subrange converting constructors permit derived-to-base slicing errors. See detailed discussion of this issue in LWG#3282.	Adopt the proposed resolution at https://cplusplus.github.io/LWG/issue3282	Accepted with Modification See LWG Issue 3282
US 286		24.06.3.2 [range.iota. view]		te	Iota_view is currently under-constrained and does not behave as a forwarding-range.	Adopt LWG 3292, and add the correct opt-in for *forwarding-range* (dependent on earlier NB comment)	Accepted with Modification See P1870
US 287		24.06.3.3		te	iota_view::iterator has the wrong iterator_category. Depending on the capabilities of the template parameter W, the category could be anything from input_iterator_tag to random_access_iterator_tag. However, according to the _Cpp17InputIterator_ requirements, iota_view::iterator cannot satisfy any of the old iterator concepts stronger than input. That is because its operator* returns a prvalue.	Adopt the proposed resolution in https://cplusplus.github.io/LWG/issue3291.	Accepted See LWG Issue 3291
DE 288		24.07		te	By fully specifying the types returned by view adaptors, the standard forces the return type of multiple chained view operations to be an increasingly nested template.	1. Adopt P1739 and in this context also P1391 and P1394 which are strongly suggested by P1739. All three papers have been seen and approved by LEWG. P1739 cannot be adopted	Accepted with Modification See P1739

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
					In general this is not avoidable, but for certain combinations of input ranges and view adaptors, one can simply create a modified object of the original type (e.g. with different bounds). Not addressing this is a design flaw that needlessly complicates working with views.	after C++20 without breaking API. 2. Discuss whether P1664 (or parts of it) should also be adopted. P1664 generalises the notion of "reconstructible ranges" (those affected by P1739). 3. Evaluate whether any other combinations of range and view adaptor should get special treatment; any such changes after C++20 are breaking.generalises the notion of "reconstructible ranges" (those affected by P1739). 4. Evaluate whether any other combinations of range and view adaptor should get special treatment; any such changes after C++20 are breaking.	
DE 289		24.07.1		te	"Given an additional range adaptor closure object D, the expression C | D is well-formed and produces another range adaptor closure object" Experience in combining range-v3 with our library's views has revealed that it is very difficult to satisfy the above requirement in a generic way since it is not defined how code can identify "range adaptor closure objects" and which entity is responsible for combining the two closures into one. This leads to incompatible implementations and conflicting overloads. It may even become impossible for developers to target different standard library implementations at the same time -- depending on how these chose to implement the above rule.	1) Introduce a boolean trait called enable_range_adaptor_closure that must be specialised for the type of all range adaptor closure objects. 2) Specify that the standard library implements the aforementioned combining of two-into-one in an implementation-defined manner (e.g. a free function operator| that works on any two objects whose types satisfy enable_range_adaptor_closure and that returns the respective combined closure object). [Note that this only affects operator| for combining two closure objects -- not for piping a range into a closure object. The trait would however also enable the standard library to provide a generic implementation of the latter so that users need only provide operator() for their closure objects. This in turn could make it possible to create closure objects	Rejected There was no consensus to adopt this change.

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
						from lambdas.]	

US 290		24.07.10		te	The strange is_reference_v<iter_reference_t<iterator_t<V>>> || view<iter_value_t<iterator_t<V>>> constraint was correct before P0970 and the *forwarding-range* concept. Now it is inexact and wordy. What we are really looking for is a *forwarding-range;* that is, a range on which we can call view::all to get a view which we can store within the join_view cheaply.	Change the requirements on the join_view class template from:
						
						template<input_range V>
						  requires view<V> &&
						input_range<range_reference_t<V>> &&
						  (is_reference_v<range_reference_t<V>> ||
						view<range_value_t<V>>)
						class join_view;
						
						to:
						
						template<input_range V>
						  requires view<V> &&
						input_range<range_reference_t<V>> &&
						  forwarding-
						range<iter_reference_t<iterator_t<V>>>
						class join_view;
							Rejected
							There was no consensus to adopt this change.

US 291		24.07.10.2		te	The non-const join_view::begin() returns iterator<simple-view<V>>. If simple-view<V> is true, then the iterator stores a const join_view* named parent_. iterator::satisfy() will try to write to parent_->inner_ if ref_is_glvalue is false. That doesn't work because the inner_ field is not marked mutable.	In [range.join.view], change the join_view<V>::inner_ member to be mutable. This is safe because this exposition-only member is only used when the join_view is single-pass and only modified by operations that invalidate other iterators. See https://cplusplus.github.io/LWG/issue3278	Accepted with Modification See P1983
US 292		24.07.10.2		te	join_view::iterator's constructor is incorrect. In join_view<V>::iterator<Const>, we see the constructor: constexpr iterator(Parent& parent, iterator_t<V> outer) V above is the non-const-qualified view template parameter. We will then try to initialize the outer_ data member with outer, which has type iterator_t<Base>, where Base is const V when Const is true, and V otherwise. This is broken; there is no required conversion if the types are different. Fixing this will probably require changes also to join_view's begin() and end() members.	In [range.join.view], change the join_view<V>::inner_ member to be mutable. This is safe because this exposition-only member is only used when the join_view is single-pass and only modified by operations that invalidate other iterators. See https://cplusplus.github.io/LWG/issue3278	Accepted with Modification See P1983

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
US 293		24.07.10.2		te	join_view is missing a base() member for returning the underlying view. All the other range adaptors provide this.	To the join_view class template add the member: constexpr V base() const { return base_; }	Accepted See LWG Issue 3322
US 294		24.07.10.3	14,15	te	join_view::iterator::operator-- is improperly constrained. In the Effects clause in paragraph 14, we see the statement: inner_ = ranges::end(*--outer_); However, this only well-formed when end returns an iterator, not a sentinel. This requirement is not reflected in the constraints of the function(s).	Change join_view::iterator::operator--() and operator--(int) to the following: constexpr iterator& operator--() requires ref_is_glvalue && bidirectional_range<Base> && bidirectional_range<range_reference_t<Base>> && common_range<range_reference_t<Base>>; constexpr iterator operator--(int) requires ref_is_glvalue && bidirectional_range<Base> && bidirectional_range<range_reference_t<Base>> && common_range<range_reference_t<Base>>;	Accepted See LWG Issue 3313
US 295		24.07.10.3	2,3	ed	Paras 2.1 and 3.2 do not say what the iterator_(category|concept) should be if neither of the two sub-bullets hold. Presumably the author intended those bullets to fall through to p2.2 and p3.3 respectively, but I don't think it works that way.	Add a 2.1.3 that reads, "Otherwise, iterator_concept denotes input_iterator_tag." Add a 3.2.3 that reads, "Otherwise, iterator_category denotes input_iterator_tag.“	Accepted - Editorial
US 296		24.07.11.3		te	split_view::outer_iterator converting constructor is slightly wrong. In split_view::outer_iterator<V, Pattern>, when V is not const-iterable, we must avoid forming the type iterator_t<const V> since it will fail to compile.	For the converting constructor: constexpr outer_iterator(outer_iterator<!Const> i) requires Const && ConvertibleTo<iterator_t<V>, iterator_t<const V>>; change the requirement to: requires Const && ConvertibleTo<iterator_t<V>, iterator_t<Base>>;	Accepted with Modification See P1983
US 297		24.07.11.4		te	The value_type of the split_view iterator is a view; however, unlike all the other view types in the ranges clause, this one does not inherit from view_interface. This is an oversight. This must be	Change the synopsis of split_view's outer_iterator to show struct split_view<V, Pattern>::outer_iterator<Const>::value_type inheriting from view_interface<value_type>.	Accepted with Modification See P1972

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
					corrected now as doing so later would change ABI.	See https://cplusplus.github.io/LWG/issue3276	
US 298		24.07.13.3	01.1	Te	The behavior of views::common cannot depend on modeling a concept.	Change "models" to "satisfies".	Accepted See P2101

GB 299		24.07.16.2		Te	has-tuple-element helper concept needs convertible_to
					The exposition-only has-tuple-element concept (for elements_view) is defined as
					template<class T, size_t N>
					concept has-tuple-element = *exposition only*
					
					requires(T t) {
					typename tuple_size<T>::type;
					requires N < tuple_size_v<T>;
					typename tuple_element_t<N, T>;
					{ get<N>(t) } -> const tuple_element_t<N, T>&;
					};
					However, the return type constraint for { get<N>(t) } is no longer valid under the latest concepts changes
						Change to:
						template<class T, size_t N>
						concept has-tuple-element = *exposition only*
						
						requires(T t) {
						typename tuple_size<T>::type;
						requires N < tuple_size_v<T>;
						typename tuple_element_t<N, T>;
						{ get<N>(t) } -> convertible_to<const tuple_element_t<N, T>&;
						};
							Accepted
							See LWG Issue 3323

US 300		24.07.2	1	Te	The behavior of semiregular-box cannot depend on modeling a concept.	Change "model{s,ed}" to "satisfie{s,d}". Add semantic constraints on the use of semiregular-box if necessary.	Accepted See P2101
GB 301		24.07.4.2		Te	filter_view has no pred() member Other views taking predicates (take_while_view and drop_while_view) have a pred() member returning (a const reference to) the contained predicate object, but filter_view does not	In [range.filter.view], add constexpr const Pred& pred() const; Effects: Equivalent to: return *pred_;	Accepted with Modification See P1983
US 302		24.07.4.2 24.7.5.2 24.7.6.2 24.7.10.2 24.7.11.2 24.7.14.2		te	Several of the view class templates in the [range.adaptors] section have converting constructors from compatible ranges. These were originally added in a misguided effort to support CTAD, but as described in LWG#3280, these constructors can cause recursion in the type constraints, leading to spurious compile errors.	Adopt the proposed resolution in https://cplusplus.github.io/LWG/issue3280	Accepted See LWG Issue 3280

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

US 303		24.07.5.2		Te	The transform_view does not constrain the return type of the transformation function. It is invalid to pass a void-returning transformation function to the transform_view, which would cause its iterators' operator* member to return void.	Change the constraints on transform_view to the following:
						
						template<input_range V, copy_constructible F>
						  requires view<V> && is_object_v<F> &&
						  regular_invocable<F&, range_reference_t<V>>
						&&
						  can-reference<invoke_result_t<F&,
						range_reference_t<V>>>
						class transform_view
							Accepted
							See LWG Issue 3286

US 304		24.07.6.2 24.5.3.1/p6		te	On an input (but not forward) range, begin(rng) is not required to be an equality-preserving expression (24.4.2 [range.range]/3.3). If the range is *also* sized, then it is not valid to call size(rng) after begin(rng) (24.4.3 [range.sized]/2.2). In several places in the ranges clause, this precondition is violated. A trivial re-expression of the effects clause fixes the problem.	Adopt the proposed resolution in https://cplusplus.github.io/LWG/issue3286	Accepted See LWG Issue 3286
FR 305		25			The range version of some algorithms are missing	Adopt P1243	Accepted with Modification See P1243
US 306		25 [algorithms]		te	The ranges comparison algorithms are overconstrained – they require symmetric comparison functions even though the algorithm doesn't need them. The constraints should be lowered. See also https://github.com/ericniebler/stl2/issues/610	Adopt P1716	Accepted See P1716
US 307		25 [algorithms]		te	Some algorithms do not have ranges:: counterparts.	Adopt P1243.	Accepted with Modification See P1243

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

GB 308		25		Te	All half ranges should be fully rangified
					It seems odd that we're not offering full ranges for the ranges that we write to. We can potentially eliminate a class of error by requiring all ranges have bounds, and implementations can optimise for the unreachable_sentinel_t case.
					This is already the case for the uninitialised memory algorithms.
						Redesign all ranges algorithms with half-ranges so that they're fully bounded.
						Example:
						// Current
						template<input_iterator I, sentinel_for<I> S, weakly_incrementable O>
						requires indirectly_copyable<I, O>
						constexpr ranges::copy_result<I, O>
						ranges::copy(I first, S last, O result);
						template<input_range R, weakly_incrementable O>
						requires indirectly_copyable<iterator_t<R>, O>
						constexpr
						ranges::copy_result<safe_iterator_t<R>, O>
						ranges::copy(R&& r, O result);
						// Proposed
						template<input_iterator I, sentinel_for<I> S1,
						input_or_output_iterator O, sentinel_for<O> S2>
						requires indirectly_copyable<I, O>
						constexpr ranges::copy_result<I, O>
						ranges::copy(I first, S1 last, O result, S2 result_last);
						template<input_range R, range O>
						requires indirectly_copyable<iterator_t<R>, iterator_t<O>>
						constexpr
						ranges::copy_result<safe_iterator_t<R>, safe_iterator_t<O>> ranges::copy(R&& r, O&& result);
							Rejected
							There was no consensus to adopt this change.

GB 309		25		Te	Strike ranges::*_n algorithms Most *_n algorithms become unnecessary in the wake of counted_iterator, and relevant range adaptors.	Remove the following ranges algorithm overloads: copy_n fill_n generate_n (Note: search_n omitted, as it appears to be different to the others.)	Rejected There was no consensus to adopt this change.
GB 310		25		Te	Some new algorithms should be marked [[nodiscard]]	Add [[nodiscard]] to the following ranges:: algorithms.	Rejected There was no consensus

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
					These algorithms are equality-preserving, and are ultimately read operations. They should be marked as [[nodiscard]] to reflect that their result is always intended to be used. Consider similar change for lexicographical_compare_three_way.	All algorithms in [alg.nonmodifying]. remove (not a read operation, but the result should rarely be discarded) remove_if (not a read operation, but the result should rarely be discarded) is_sorted is_sorted_until All algorithms in [alg.binarysearch]. is_parititoned partition_point includes is_heap All algorithms in [alg.min.max]. lexicographical_compare Possibly also add nodiscard to lexicographical_compare_three_way.	to adopt this change.
CZ 311		25	24.5	te	Due to the ranges API being more or less fixed after shipment, API-breaking fixes have to be scheduled now rather than shipped in C++23. Currently, many of the new algorithms in std::ranges algorithms copy their boolean-returning predecessors by returning a single boolean value. And while this makes perfect sense as an independent unit, individuals composing these algorithms with iterators that are bidirectional or worse sometimes need to perform additional actions around and because of the return of one of these boolean-returning algorithms. For example, if an iterator is advanced to its corresponding "last" value by std::equal, that advancement is lost upon returning just a boolean from the algorithm. Any work that wanted to continue from the "last" value supplied into the algorithm must re-increment the iterator, resulting in duplicated work.	The algorithms (the new ones in std::ranges) should be changed to have a result type which is (explicitly) convertible to boolean and also retains the Iterator value at its state. If the algorithm is "successful" (e.g., std::equal returns true), the iterator must point at the end. Otherwise, the value of the returned iterator is unspecified. Purportedly, an upcoming paper P1877 will handle this.	Rejected There was no consensus to adopt this change.
PL 312		25 [algorithms]		te	The ranges::is_permutation, ranges::unique, ranges::unique_copy algorithm are currently	Adopt P1716R2.	Accepted See P1716

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
					underconstrained, as they do not require supplied functor to model equivalence relation. Other compare algorithms (like ranges::equal, ranges::mismatch, ranges::search), are overconstrained. They require supplied functor to model relation<T, U> concept instead of predicate<T, U>, where T and U are reference types of supplied ranges. As consequence, supplied functor needs to be callable with four combinations of arguments: (T, U), (U, T), (U, U), (U, U), instead of just (T, U). This makes them less general than existing non-range overloads, and complicates code migration.		
US 313		25.02 [algorithms.requirements]		Ge	This subclause describes general purpose wording that applies to all algorithms in the standard, without defining algorithm. It generally applies such definitions to algorithms "in this clause", but the wording for specialized algorithms in 20.10.11 [specialized.algorithms] relies on this wording too, especially to provide definitions for its ranges overloads.	Revise this subclause to include a definition of *algorithm*, so that all the wording that applies to this subclause instead applies to all algorithms. Reasonable definitions of algorithm (for the purposes of the standard library) might be all function templates in clause 25, and clause 20.10.11, or some definition constructed around function templates having arguments of iterator or range type. The former is likely a simpler fix for C++20, the latter would avoid having to update the list of locations for algorithms in the future,	Accepted with Modification See P1963
JP8 314		25.03.1		ed	Notes in terminological entries should start with different element, namely, "Note # to entry", according to Clauses 24 and 16.5.9 in the Directives Part 2.	Replace "[Note:" with "[Note 1 to entry:".	Accepted - Editorial

GB 315		25.04		Te	next_permutation_result has no conversion operators
					The other *_result classes have conversion operators defined (for the case where the range-based overload returns dangling) but next_permutation_result does not. It is also missing the [[no_unique_address]] attribute for its iterator member.
						In [algorithm.syn], change
						template<class I>
						struct next_permutation_result {
						bool found;
						[[no_unique_address]] I in;
						template <class I2>
						requires convertible_to<const I&, I2>
						operator next_permutation_result<I2>() const & {
						return {found, in};
						}
						template <class I2>
						requires convertible_to<I, I2>
						operator next_permutation_result<I2>() && {
							Accepted with Modification
							See P2106

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

						return {found, std::move(in)};
						}
						};	

GB 316		25.04		Te	Algorithm result types should be distinct types; not aliases
					Each algorithm should have its own result type (that might be derived from some common exposition-only type).
					It will be probably be confusing for a diagnostic to report copy_result as the return type when a user is using move_backward, or for mismatch_result to appear when a user is using swap_ranges!
						Add three exposition-only types:
						template<class I1, class I2>
						struct in1-in2-result {
						[[no_unique_address]] I1 in1;
						[[no_unique_address]] I2 in2;
						template<class II1, class II2>
						requires convertible_to<const I1&, II1> &&
						convertible_to<const I2&, II2>
						operator in1-in2-result<II1, II2>() const & {
						return {in1, in2};
						}
						template<class II1, class II2>
						requires convertible_to<I1, II1> &&
						convertible_to<I2, II2>
						operator in1-in2-result<II1, II2>() && {
						return {std::move(in1), std::move(in2)};
						}
						};
						
						template<class I, class O>
						struct in-out-result {
						[[no_unique_address]] I in;
						[[no_unique_address]] O out;
						template<class I2, class O2>
						requires convertible_to<const I&, I2> &&
						convertible_to<const O&, O2>
						operator in-out-result<I2, O2>() const & {
						return {in, out};
						}
						template<class I2, class O2>
						requires convertible_to<I, I2> &&
						convertible_to<O, O2>
						operator in-out-result<I2, O2>() && {
						return {std::move(in), std::move(out)};
						}
						};
						
						template<class I1, class I2, class O>
							Accepted with Modification
							See P2106

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

						struct in1-in2-out-result {
						[[no_unique_address]] I1 in1;
						[[no_unique_address]] I2 in2;
						[[no_unique_address]] O out;
						template<class II1, class II2, class OO>
						requires convertible_to<const I1&, II1> &&
						convertible_to<const I2&, II2> &&
						convertible_to<const O&, OO>
						operator in1-in2-out-result<II1, II2, OO>()
						const & {
						return {in1, in2, out};
						}
						template<class II1, class II2, class OO>
						requires convertible_to<I1, II1> &&
						convertible_to<I2, II2> && convertible_to<O,
						OO>
						operator in1-in2-out-result<II1, II2, OO>() &&
						{
						return {std::move(in1), std::move(in2),
						std::move(out)};
						}
						};
						Each of the algorithm_result types should be privately derived from the relevant exposition-only type, with the members being made publicly available.
						next_permutation_result should be renamed to permutation_result. (Note that for_each_result, partition_copy_result, minmax_result, and next_permutation_result don't have an exposition-only type, since their use-cases are mostly unique.)	

FR 317		25.06.14		te	The names of shift_left and shift_right will be misleading when specialized for bit proxy iterators (shift_left will call the bitwise right shift operator >>, and shift_right will call the bitwise left shift operator <<). As a result, the names of the algorithms shift_left and shift_right would benefit from being adjusted to less misleading names.	Alternatives include having only one shift algorithm taking a signed integer for the shift amount, and shifting to the beginning for a negative amount, and to the end for a positive amount. Changing names is another alternative: shift_next/shift_prev, backshift/foreshift, back_shift/fore_shift, shift_forward/shift_backward, shift_front/shift_back,	Rejected There was no consensus to adopt this change.

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
						shift_begin/shift_end	
US 318		25.06.14 [alg.shift]		te	Currently, shift by a negative value is simply ignored. This runs totally counter to user expectation. Make it a precondition to provide a non-negative value. This would be a behaviour change so needs to be done now.	Adopt P1243. If not that, adopt P1233.	Accepted with Modification See P1243
JP9 319		25.07.8	p22	ed	"Let X be the return type. Returns Xx, y, where ..." needs braces for constructing the value. In addition, other descriptions for "Returns" doesn't specify the return type explicitly. It would be better to make consistent.	{x, y}, where ...	Accepted - Editorial
US 320		25.08 [numeric. ops. overview]		te	We made lots of algorithms constexpr, but not the ones in <numeric>. We really should be more thorough and not just forget these.	Adopt P1645.	Accepted See P1645.
FR 321		25.09			All non-allocating algorithms have been made constexpr except for the ones in numeric, which seems like an oversight	Adopt P1645	Accepted See P1645.
PL 322		25.09 [numeric.ops]		te	● The specification of GENERALIZED_*_SUM is overly restrictive and suggests that a serial cutoff is not allowed. ● The intermediate type for numeric algorithms is unclear. ● The type requirements for numeric algorithms	Adopt P0571.	Rejected There was no consensus to adopt this change.

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
					are unclear. ● The requirements on function objects for numeric algorithms are overly restrictive.		
FR 323		26.05		te	The current wording of low level bit manipulation functions specified by "P0553R4: Bit operations" and by "P0556R3: Integral power-of-2 operations" make these functions unusable with std::byte. Since the whole purpose of the introduction of std::byte was to break the ambiguity between pure integers vs memory storage, the incompatibility between low level bit functions and std::byte reintroduces this pre-C++17 ambiguity for users. The current design may make future evolution of the <bit> header more complicated.	Introduce a machine word “concept” as well as related type traits (binary_digits, is_word) that unsigned integers, extended unsigned integers, and std::byte satisfy. This mechanism should also constitute a customization point for advanced users who want to provide their own words types. Have the low-level bit operations take machine words as inputs and not only unsigned integers. See paper P1856R1.	Rejected There was no consensus to adopt this change.
US 324		26.05 Applies to §26.5 Numerics Library, Bit manipulation [bit]	§26.5.4 Integral powers of 2 [bit.pow.two]; §26.5.5 Rotating [bit.rotate]; §26.5.6 Counting [bit.count]	te	By strict interpretation of the wording, none of these new bit-oriented interfaces work with std::byte. Given the discussion in the minutes it seems like this was something intentionally left out, to be patched later. A cast to unsigned integral type for std::byte makes a type which already suffers from lack of math operations and similar even more verbose when working with operations it is absolutely supposed to apply to. This is not a useful restriction.	For sections §26.5.4 Integral powers of 2 [bit.pow.two]; §26.5.5 Rotating [bit.rotate]; §26.5.6 Counting [bit.count], change the "Remarks" text to be as follows -- "Remarks: This function shall not participate in overload resolution unless T is an unsigned integer type ([basic.fundamental]) or byte ([cstddef.syn]). If T is byte, then the expression is equivalent to std::???(static_cast<underlying_type_t<byte>>(value)) ." Substitute the name of each function from the section for ???.	Rejected There was no consensus to adopt this change.
US 325		26.05 [bit]		Ed	The contents of the header <bit> relate to inspecting and manipulating memory patterns directly, rather than numeric operations. It better belongs under clause 20, general utilities.	Move 26.5 [bit] to a new subsection under class 20 [utilities]	Rejected There was no consensus to adopt this change.
PL 326		26.05 [bit]		te	The name of std::log2p1 clashes with an IEEE-754 algorithm of the same name, which has been included in a C TS meant for inclusion in a future C standard (and probably also in C++). There is no ambiguity between the overloads, however having a name overloaded for two completely different mathematical formulas is not a good thing. For reference, the log2p1 algorithm from the CD is `log2p1(x) = (x == 0) ? 0 : 1 + floor(log2(x))`, while the log2p1 algorithm from	Rename `std::log2p1` to `std::bit_length`.	Accepted with Modifications See P1956

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
					IEEE-754 `log2p1(x) = log2(1 + x)`. Additionally, the function the CD calls log2p1 is commonly - at the very least in Python, Ruby, Java, and Dart - called `bit_length`. To the author of this comment, it seems preferable to follow the name that is used in other languages, to make it easier for programmers to communicate and reason about code across the different languages.		
US 327		26.05.4 [bit.pow.two]		ed	`log2p1` collides with an IEEE-754 operation.	Rename `log2p1`. We suggest a different semantically meaningful name such as `bit_width` or `base2digits`.	Accepted with Modifications See P1956
US 328		26.05.4 [bit.pow.two]		ed	`ceil2` and `floor2`'s names are unintuitive. Meaning that most programmers reading the code won't know what's meant.	Rename `ceil2` and `floor2`.	Accepted with Modifications See P1956
US 329		26.05.4 [bit.pow.two]		te	The behavior of `ceil2` and `floor2` at 0 is unlikely to be something programmers use correctly.	Change the constraints for both functions around 0.	Rejected There was no consensus to adopt this change.
US 330		26.05.4 [bit.pow.two]		ed	The specification says: "The minimal value `y` such that `ispow2(y)` is `true` and `y >= x`; if `y` is not representable as a value of type `T`, the result is an unspecified value.“ `y` is an argument to `ispow2(y)`. It is necessarily representable. Note - the above words are not in the CD, SC22 N5410 (WG21 N4830). See 26.5.4	Rephrase. Better wording might be "if no such `y` exists", but that doesn't seem particularly useful: how does one differentiate "no such `y`" from a real answer?	Rejected There was no consensus to adopt this change.
GB 331		26.05.4	3,4,5,6	Te	std::ceil2() & std::floor2() produce conceptual confusion std::ceil() is a linear operation, but std::ceil2() is an exponential operation. The '2' suffix does not provide any hint as to its fundamental difference from the std::ceil() function. std::floor2() suffers from the same defect. Spell out what the operations are exactly by renaming to ceil_power_of_two() and floor_power_of_two().	Rename std::ceil2() to std::ceil_power_of_two(). Rename std::floor2() to std::floor_power_of_two().	Accepted with Modifications See P1956

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
GB 332		26.05.4	7, 8	Te	std::log2p1() from P0556 introduces a possible name collision. It is defined as: log2p1(x) = (x == 0) ? 0 : 1 + floor(log2(x)) In IEEE754-2008, and WG14 TS 18661-4a (targeted for C2X): log2p1(x) = log2(1+x) The intention of P0556's log2p1() function is to facilitate bit manipulation algorithms by computing the number of bits needed to represent an unsigned integer. Give it a more descriptive name, such as std::bits_needed().	Rename std::log2p1() to std::bits_needed().	Accepted with Modifications See P1956
GB 333		27.02.2.1		Te	UTC epoch is not correctly defined UTC has an officially recorded epoch of 1/1/1972 00:00:00 and is 10 seconds behind TAI. This can be confirmed through reference to the BIPM (the body that oversees international metrology) https://www.bipm.org/cc/CCTF/Allowed/18/CCTF_09-32_noteUTC.pdf Specifically page 6 "The defining epoch of 1 January 1972, 0 h 0 m 0 s UTC was set 10 s behind TAI, accumulated difference between TAI and UT1 since the inception of TAI in 1958, and a unique fraction of a second adjustment was applied so that UTC would differ from TAI by an integral number of seconds. The recommended maximum departure of UTC from UT1 was 0.7 s. The term “leap second” was introduced for the stepped second."	utc_clock and utc_timepoint should correctly report relative to the official UTC epoch. 27.2.2.1 footnote 1 should read In contrast to sys_time, which does not take leap seconds into account, utc_clock and its associated time_point, utc_time, count time, including leap seconds, since 1972-01-01 00:00:00 UTC. [Example: clock_cast<utc_clock>(sys_seconds{sys_days{1972y/January/1}}).time_since_epoch() is 0s. clock_cast<utc_clock>(sys_seconds{sys_days{2000y/January/1}}).time_since_epoch() is 883'612'822, which is 10’197 * 86’400s + 22s. — end example]	Accepted See LWG Issue 3316
US 334		27.05.10 [time. duration.io]		te	operator<< for floating-point durations always produces output with six digits after the decimal point, and doesn’t use the stream’s locale either.	Rewrite the specification to not rely on to_string() for floating-point formatting.	Accepted See LWG Issue 3317
GB 335		27.07.1.1			Wording for clocks should be unified unless they are intended to behave differently	Unify the wording	Accepted See LWG issue 3318

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
					In 27.7.1.1 note 1 for system_clock it is stated "Objects of type system_clock represent wall clock time from the system-wide realtime clock. Objects of type sys_time<Duration> measure time since (and before) 1970-01-01 00:00:00 UTC" The express statement of "since (and before)" is important given the time epoch of these clocks. If all the clocks support time prior to their zero-time then this should be stated explicitly. If not then likewise that should be noted. No change is proposed yet, clarification required over the intended behaviour when using values prior to a given clock's epoch is needed before the appropriate change can be suggested.		
GB 336		27.07.4		Te	Use of specific clocks may create expectations that are not which was the approximate delivered (GPS) The "gps" clock has nothing to do with the GNSS service known as GPS except for sharing a common anchor point (epoch) there is no calibration, no feed, no expectation that the "clock" correlates to GPS data streams. It seems a very niche use and given some of the other issues around its interpretation I would suggest it is removed.	delete gps_clock	Rejected There was no consensus to adopt this change.
GB 337		27.07.4		Te	gps_clock is a unilateral reference to a US service and has no place alone in the ISO standard The GPS GNSS service is owned, maintained and controlled by the US government and while Satellite timing and position usage has become all but ubiquitous in many applications a modern GNSS receiver is capable of receivng updates from multiple GNSS constellations to ensure coverage and security. If GPS is represented then other national and interenationally maintained services should be included, Glonass, Beidou, Galileo to name but 3. Each of these	Remove gps_clock	Rejected There was no consensus to adopt this change.

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
					have different operating parameters, most notably the epoch and the application or not of leap seconds. https://gssc.esa.int/navipedia/index.php/Time_References_in_GNSS While Galileo and Beidou do not respect leap seconds, GLONASS does and GLONASS transmits at a constant offset of 3 hours relative to UTC (being Russian standard time). Other features of gps, such as the week rollover (an epoch defining event and which occurs every 19 years, the most recent being this April 2019) are not represented in the gps_clock. If one purpose of providing a gps_clock is to allow the comparison of gps data to other clocks then epoch rollover probably ought to be recognised, though hopefully it will become a thing of the past as increasingly gps satellites are upgrading to a large week counter)		
JP10 338		27.08.3.3	p10	ed	This is different from the declaration in 27.2.	constexpr chrono::day operator""d(unsigned long long d) noexcept;	Accepted – Editorial
JP12 339		27.08.4.2	p14	ed	Class name is not required.	constexpr explicit ~~month::~~operator unsigned() const noexcept;	Accepted – Editorial
JP13 340		27.08.4.2	p15	ed	Class name is not required.	constexpr bool ~~month::~~ok() const noexcept;	Accepted – Editorial
JP11 341		27.08.4.2	p2	ed	Class name is not required.	constexpr month& ~~month::~~operator++() noexcept;	Accepted – Editorial
GB 342		27.08.4.3		Te	std::chrono::month is the only duration-like type without a UDL, which makes constructing objects such as this look a little off. chrono::year_month_day{1815y, chrono::month{12}, 10d}}	Add operator""month, which behaves similarly to operator""y. The earlier expression could then read as:	Rejected There was no consensus to adopt this change.

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
						chrono::year_month_day{1815y, 12month, 10d}}	
JP14 343		27.08.5.3	p10	ed	This is different from the declaration in 27.2.	constexpr chrono::year operator""y(unsigned long long y) noexcept;	Accepted – Editorial
DE 344		27.11.01	paragraph 1	ge	This paragraph says "27.11 describes an interface for accessing the IANA Time Zone database described in RFC 6557, ..." However, RFC 6557 does not describe the database itself; it only describes the maintenance procedures for that database, as its title implies (quoted in clause 2).	Add a reference to a specification of the database itself, or excise all references to the IANA time zone database.	Accepted with Modification Reference to IANA time zone database.
DE 345		27.11.08		te	The class name "leap" to designate a UTC leap second event is too generic and not sufficiently descriptive.	Rename the class to "utc_leap" or "utc_leap_second", Consistent with the naming of "utc_clock" and "utc_time" for other UTC-related classes.	Accepted with Modification See P1981
DE 346		27.11.09		te	The class name "link" to designate an alias for a named time zone is too generic and not sufficiently descriptive.	Rename the class to "zone_link", consistent with the fact that all other classes related to time zones contain "zone" in their name.	Accepted with Modification See P1982
FR 347		27.12.10			local_time_format should use optional<string> and optional<seconds> instead of pointers	change local_time_format signature to local_time_format(local_time<Duration> time, optional<string> = {}, optional<seconds> offset_sec = {});	Rejected There was no consensus to adopt this change.
JP15 348		29.10.01		ed	The default template arguments are missing.	template<class charT, class traits = char_traits<charT>, class Allocator = allocator<charT>>	Accepted – Editorial

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
						class basic_syncbuf; template<class charT, class traits = char_traits<charT>, class Allocator = allocator<charT>> class basic_osyncstream;	
JP16 349		29.10.02.1		ed	The default template arguments are missing.	template<class charT, class traits = char_traits<charT>, class Allocator = allocator<charT>> class basic_syncbuf : public basic_streambuf<charT, traits> {	Accepted – Editorial
JP17 350		29.10.03.1		ed	The default template arguments are missing.	template<class charT, class traits = char_tratis<charT>, class Allocator = allocator<charT>> class basic_osyncstream : public basic_ostream<charT, traits> {	Accepted – Editorial
US 351		31 [atomics]		te	Atomic initialization has been broken since C++11.	Adopt P0883R1.	Accepted with Modification See P0883
US 352		31 [atomics]		te	It is not possible to include C's <stdatomic.h> in C++ today, which makes it difficult to use atomics in code that needs to be compiled as both C and C++. C++ should support inclusion of <stdatomic.h>.	Adopt P0943.	Rejected There was no consensus to adopt this change.
CA 353		31 [atomics]		te	Atomic initialization doesn't work as expected.	Adopt P0883R1.	Accepted with Modification See P0883
US 354		31.03 [atomics. alias]		Te	In P1135r3, the atomic_int_fast_wait_t and atomic_uint_fast_wait_t type aliases were removed. The paper's changelog explains why: > Removed atomic_int_fast_wait_t and atomic_uint_fast_wait_t, because LEWG at San Diego 2018 felt that the use case was uncommon and the types had high potential for misuse. We think this decision warrants reconsideration. On some platforms, certain implementation strategies for wait/notify are only available for certain sized integer types (for example, Linux's futex is for int only)	Re-add P1135r2's atomic_int_fast_wait_t and atomic_uint_fast_wait_t.	Rejected There was no consensus to adopt this change.

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
US 355		31.07 [atomics.ref.generic] 31.7.1 [atomics.ref.operations] 31.7.2 [atomics.ref.int] 31.7.3 [atomics.ref.float] 31.7.4 [atomics.ref.pointer]	25-28 1 1 1	te	atomic_ref<T>::notify_one and atomic_ref<T>::notify_all should be const member functions (in the generic class and all the specializations), since it is the atomic object that is (conceptually) modified, not the atomic_ref<T> object.	Make atomic_ref<T>::notify_one and atomic_ref<T>::notify_all const.	Accepted with Modification See P1960
US 356		31.07.1 [atomics.ref.operations]	10	te	atomic_ref::is_lock_free should require that the result only depend on the type of the object, not the specific object. The current specification is inconsistent with atomic<T>. (See [atomics.lockfree] 31.5p3.) This test is primarily useful to determine whether a particular algorithm can or should be used. If the result can vary based on object identity, that is not possible. There is no way to ask whether the property holds for all relevant objects until all of the objects are actually available for testing. Note that is_always_lock_free does not fully serve this purpose, since is_lock_free() may vary at run time depending on hardware characteristics, while still being consistent per type. This was the subject of recent reflector discussion.	Apply the PR for LWG3249, and replicate the equivalent wording here. Possibly consider making the is_lock_free member function static. That appears to make the member function behavior less surprising, at the cost of an inconsistency with the C-constrained free function.	Accepted with Modification See P1960
US 357		31.07.2 [atomics.ref.int] 31.8.2 [atomics.	1 1	ed	The note at the end of [atomics.ref.int] (31.7.2) paragraph 1: [*Note*: For the specialization atomic_ref<bool>, see 31.7. — *end note*] refers to [atomics.ref.generic](31.7). There is no mention of atomic_ref<bool> in that subclause,	Change the note at the end of [atomics.ref.int] (31.7.2) paragraph 1 as follows: [*Note*: For the specialization atomic_ref<bool>, see 31.7. — *end note*] [*Note*: The specialization atomic_ref<bool> is based on the primary template	Accepted - Editorial

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
		types.int]			so the reason for the cross reference is not obvious. The note at the end of [atomics.types.int] (31.8.2) paragraph 1: [*Note*: For the specialization atomic<bool>, see 31.8. -- *end note*] has a similar issue. [atomics.types.generic] (31.8) does mention atomic<bool>, but not in a way that makes the reason for the cross reference obvious.	([atomic.ref.generic]), and is not included among the integer-specific specializations. — *end note*] Change the note at the end of [atomics.types.int] (31.8.2) paragraph 1 as follows: [*Note*: For the specialization atomic<bool>, see 31.8. -- *end note*] [*Note*: The specialization atomic<bool> is based on the primary template ([atomics.types.generic]), and is not included among the integer-specific specializations. -- *end note*]	
US 358		31.07.3 [atomics.ref. float]	1	te	In the atomic_ref<*floating-point*> synopsis in [atomic.ref.float] (31.7.3) paragraph 1: *floating-point* operator=(*floating-point*) noexcept; should be a const member function, like all other atomic_ref<T> assignment operators.	Change the atomic_ref<*floating-point*> synopsis in [atomic.ref.float] (31.7.3) paragraph 1 as follows: *floating-point* operator=(*floating-point*) const noexcept;	Accepted with Modification See P1960
US 359		31.07.5 [atomics.ref. memop]	1-4	ed	In the specification of member operations common to atomic_ref<*integral*> and atomic_ref<T*> specializations in [atomics.ref.memop] (31.7.5), all of the member functions are specified to return T*, which is only correct for the atomic_ref<T*>. The corresponding member operations for atomic<*integral*> and atomic<T*> in [atomics.types.memop] (31.8.5) return *T*. Additionally, there is an extra int parameter in the specification of the second operator--. That declaration is supposed to be the predecrement operator, not the postdecrement operator.	Change the specification of member operations common to atomic_ref<*integral*> and atomic_ref<T*> specializations in [atomics.ref.memop] (31.7.5) as follows: T* operator++(int) const noexcept; *Effects:* Equivalent to: return fetch_add(1); T* operator--(int) const noexcept; *Effects:* Equivalent to: return fetch_sub(1); T* operator++() const noexcept; *Effects:* Equivalent to: return fetch_add(1) + 1; T* operator--(int) const noexcept; *Effects:* Equivalent to: return fetch_sub(1) - 1; Alternatively, we could consider rewording these member operations entirely for both atomic<T> and atomic_ref<T>. For example, we could just add them to both the integer and pointer specializations, which would be clearer, but would duplicate the wording.	Accepted with Modification See P1960
GB		32		Te	Too many new headers	Consolidate them into a single <sync> header or similar.	Rejected

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
360					The CD includes new headers <stop_token>, <semaphore>, <latch>, and <barrier>. It seems unhelpful to split the synchronization utilities across so many different headers.		There was no consensus to adopt this change.
GB 361		32.04.3		Te	Rename jthread The name jthread, while concise, is cryptic. By expanding the j, the 'smart-thread' type will be in sync with its smart-pointer cousins.	Proposed Change: Consider expanding jthread into a name that more directly represents its intention. Examples include, but are not limited to: joining_thread join_thread	Rejected There was no consensus to adopt this change.
JP18 362		32.04.3.5		ed	This is different from the declaration in 32.4.3.	[[nodiscard]] unsigned hardware_concurrency() noexcept;	Accepted - Editorial
PL 363		32.06.4 [thread.condition.condvarany]		te	The conditition_variable_any::wait_until that accepts lock and stop_token, is inconsistent with the [thread.req.timing] p4, that reserves '_until' suffix for functions that accepts time_point. Furthermore, all interruptible waits functions, are accepting stop_token as the last argument, following the predicate, thus making them harder to format in situations when lambda is passed as a predicate.	Change the interruptible waits interface as proposed in P1869R0.	Accepted with Modification See P1869
US 364		32.07.2	Paragraph 13	te	The phrasing of the spurious failure case of semaphore try_acquire can confuse readers, who may parse it as being about blocking guarantees or a statement about QoI, rather than capturing various memory model subtleties as intended. Better would be to word this case similarly to mutex try_lock. The proposed change does so.	Replace with "Effects: Attempts to atomically check if the counter is positive and decrement it by one if so, without blocking. If the counter is not decremented, there is no effect and try_acquire immediately returns. An implementation by fail to decrement the counter even if it is positive. [Note: This spurious failure is normally uncommon, but allows interesting implementations based on a simple compare and exchange ([atomic]). -- end note] An implementation should ensure that try_acquire() does not consistently return false in the absence of contending semaphore operations.“	Accepted with Modification See P1960
US 365		32.08 [thread.coord]		te	latch and barrier currently take a ptrdiff_t as their expected count parameter and thus must support any expected count (larger than or equal to 0) that will fit in a ptrdiff_t. This limits implementation freedom; some platforms can provide a much	Adopt P1865, which adds a static constexpr ptrdiff_t max() noexcept; member to both classes that returns the expressible range of the object, like the one on counting_semaphore.	Accepted with Modification See P1865

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
					more efficient implementation of latch and barrier if they can restrict the maximum possible expected count.		
CA 366		32.08.1 [thread.latch]		ed	Subclauses 32.8.2 [latch.syn] and 32.8.3 [thread.latch.class] should be subclauses of subclause 32.8.1 [thread.latch] instead of at the same level (e.g., to be consistent with subclause 32.8.4 [thread.barriers]).	Make subclauses 32.8.2 and 32.8.3 subclauses of 32.8.1.	Accepted - Editorial
US 367		6-15		ge	Requirements that a header be included before a language feature is functional should also allow for importing that header (unit), While we will file additional comments for the cases we identify, this is a catch-all comment to adopt the principle and similarly fix any places we miss.	When a header be included before a language feature is functional should also allow for importing that header (unit).	Accepted with Modification See P1971
US 368		6-15		ge	Undefined behavior lexing and tokenizing the program text has no place in a modern standard. Unnecessary undefined behaviour in our standard raises a wide variety of concerns, not least with the security community, and all concerns related to turning source code into tokens for subsequent analysis should be either diagnosable errors, or (conditionally) supported behavior. This comment is a principle statement for more detailed comments that follow, and as a catch-all for any further occurrences that were missed.	Remove Undefined Behavior lexing and tokenizing the program text.	Rejected There was no consensus to adopt this change.
BG3 369	P 118	7.06.2.3	6	ge	(Related to BG2) The code example uses the void-returning variant of await_suspend().	Change the return type of my_future::await_suspend() and awaiter::await_suspend() to coroutine_handle.	Rejected There was no consensus to adopt this change.
US 370		All clauses affected by P0912R5 Beh ??? 9.04.4 et al	n/a	ge	WG21 has received four independent usage reports on efforts to adopt Coroutines in production code: P0054, P0973, P1471, and P1745. All of these early adopters identified major problems that could not be fixed in a backwards-compatible way; the problems identified in P0054 were addressed via incompatible changes, and the problems in the other papers remain unaddressed in the CD. On the basis of this experience, we believe it would	Revert the application of P0912R5 (Merge Coroutines TS into C++20 working draft).	Rejected There was no consensus to adopt this change. CHECK THIS

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
					be premature to standardize Coroutines.		
RU 371		All the library clauses		ge	Apply all the wordings from all the “Mandating the Standard Library” papers.	Apply wordings from P1505, P1622, P1686, P1718-P1723.	Accepted See P1505 is superceeded by P1723. P1622 P1686 P1718 P1719 P1720 P1721 P1722 Are all accepted.
US 372		Annex C [c.compat]		te	C and C++ atomics haven't worked together properly since first being standardized, even if the intent was for them to interoperate.	Adopt P0943R4.	Rejected There was no consensus to adopt this change.
JP19 373		C.5.01	p2.6	ed	It's good to have a reference as in p2.1 to p2.5	Add a reference to 7.5.7.	Accepted - Editorial
NL 374	3	Cross references from ISO C ++ 2017"		ed	Typo ‘fmtflatgs’ should be ‘fmtflags’.	Change to ‘fmtflags’	Accepted - Editorial

NL 375	2	D.19 [depr.fs.path.factory]	Sub 4	te	Example in deprecated section implies that std::string is the type to use for utf8 strings.
					[Example: A string is to be read from a database that is encoded in UTF-8, and used to create a directory using the native encoding for filenames:
					namespace fs = std::filesystem;
					std::string utf8_string = read_utf8_data();
					fs::create_directory(fs::u8path(utf8_string));
						Add clarification that std::string is the wrong type for utf8 strings	Accepted
							See LWG Issue 3328

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
JP20376		D.7	p1	ed	9.4 is less appropriate as a reference for "these implicit definitions could become deleted"	Replace 9.4 with 9.4.3.	Accepted - Editorial
CA377		General		te	C / C++ interop for atomics is buggy.	Adopt P0943.	Rejected There was no consensus to adopt this change.
CA378		General		ge	How constraints work with non-templated functions is still under heavy construction during this late stage in the process. While we have provided various comments that build in a direction where supporting such constructs (including ordering between multiple constrained functions based on their constraints) would become possible, we acknowledge that WG 21 might not find a solution with consensus in time for the DIS. We ask WG 21 to evaluate the risk of shipping the feature in such a state and consider removing the ability to declare such functions.		Accepted with Modification See P1971

Programming languages -- 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