ISO/IEC JTC1 SC22 WG21 P0636R0
Date: 2017-04-02
To: the public
Thomas Köppe <tkoeppe@google.com>

Changes between C++14 and C++17 DIS

Abstract

This document enumerates all the major changes that have been applied to the C++ working draft since the publication of C++14, up to the publication of the C++17 DIS (N4660). Major changes are those that were added in the form of a dedicated paper, excluding those papers that are large issue resolutions. No issue resolutions from either CWG or LWG issues lists ("defect reports") are included.

Contents

Removed or deprecated features

Document	Summary	Examples, notes
N4086	Remove trigraphs	The sequence `??!` no longer means `|`. Implementations may offer trigraph-like features as part of their input encoding.
P0001R1	Remove `register`	The `register` keyword remains reserved, but it no longer has any semantics.
P0002R1	Remove `++` for `bool`	Increment (`++`) prefix and postfix expressions are no longer valid for operands of type `bool`.
P0003R5	Remove `throw(A, B, C)`	Dynamic exception specifications of the form `throw(A, B, C)` are no longer valid. Only `throw()` remains as a synonym for `noexcept(true)`. Note the change in termination semantics.
P0386R2	Deprecate redeclaration of static constexpr class members	Given `struct X { static constexpr int n = 10; };`, `int X::n;` is no longer a definition, but instead a redundant redeclaration, which is deprecated. The member `X::n` is implicitly inline (see below).
N4190	Remove `auto_ptr`, `random_shuffle`, old parts of `<functional>`	Features that have been deprecated since C++11 and replaced with superior components are no longer included. Their names remain reserved, and implementations may choose to continue to ship the features.
P0004R1	Remove deprecated iostream aliases	Same as above

Document	Summary	Examples, notes
P0302R1	Remove allocator support from `function`	The polymorphic function wrapper `function` no longer has constructors that accept an allocator. Allocator support for type-erasing, copyable types is difficult, and possibly not implementable efficiently.
P0063R3 (see below)	Deprecate C library headers	The following headers of the "C library" (this is the term for a part of the C++ standard library, *not* a part of the C standard!) are now deprecated: `<ccomplex>`, `<cstdalign>`, `<cstdbool>`, `<ctgmath>`. Note that the header `<ciso646>` is not deprecated.
P0174R2	Deprecate old library parts	These library components are now deprecated: `allocator<void>`, `raw_storage_iterator`, `get_temporary_buffer`, `is_literal_type`, `std::iterator`.
P0618R0	Deprecate `<codecvt>`	The entire header `<codecvt>` (which does *not* contain the class `codecvt`!) is deprecated, as are the utilities `wstring_convert` and `wbuffer_convert`. These features are hard to use correctly, and there are doubts whether they are even specified correctly. Users should use dedicated text-processing libraries instead.
P0371R1	Deprecate `memory_order_consume` temporarily	The current semantics of "consume" ordering have been found inadequate, and the ordering needs to be redefined. While this work is in progress, hopefully ready for the next revision of C++, users are encouraged to not use this ordering and instead use "acquire" ordering, so as to not be exposed to a breaking change in the future.
P0521R0	Deprecate `shared_ptr::unique`	This member function suggests behaviour that is not actually provided.
P0604R0	Deprecate `result_of`	Use the new trait `invoke_result` instead.

New core language features with global applicability

These are features that may happen to you without your knowledge or consent.

Document	Summary	Examples, notes
P0012R1	Exception specification as part of the type system	The exception specification of a function is now part of the function's type: `void f() noexcept(true);` and `void f() noexcept(false);` are functions of two distinct types. Function pointers are convertible in the sensible direction. (But the two functions `f` may not form an overload set.) This change strengthens the type system, e.g. by allowing APIs to require non-throwing callbacks.
P0135R1	Guaranteed copy elision	The meaning of *prvalue* and *glvalue* has been revised, prvalues are no longer objects, but merely "initialization". Functions returning prvalues no longer copy objects ("mandatory copy elision"), and there is a new prvalue-to-glvalue conversion called *temporary materialization conversion*. This change means that copy elision is now guaranteed, and even applies to types that are not copyable or movable. This allows you to define functions that return such types.

Document	Summary	Examples, notes
P0035R4	Dynamic allocation of over-aligned types	Dynamic allocation (`operator new`) may now support over-aligned types, and a new overload of the operator takes an alignment parameter. It is still up to the implementation to choose which alignments to support.
P0145R3	Stricter order of expression evaluation	The order of evaluation of certain subexpressions has been specified more than it used to be. An important particular aspect of this change is that function arguments are now evaluated in an indeterminate order (i.e. no interleaving), which was previously merely unspecified. Note that the evaluation order for overloaded operators depends on how they are invoked: when invoked using operator syntax, the order is the same as for the built-in operator, but when invoked using function call syntax, the order is the same as for ordinary function calls (i.e. indeterminate).

New core language features with local applicability

These are features where you would know if you were using them.

Document	Summary	Examples, notes
N4267	A `u8` character literal	A character literal prefix `u8` creates a character that is a valid Unicode code point that takes one code unit of UTF-8, i.e. an ASCII value: `u8'x'`
P0245R1	Hexadecimal floating point literals	Floating point literals with hexadecimal base and decimal exponent: `0xC.68p+2`, `0x1.P-126`. C has supported this syntax since C99, and `printf` supports it via `%a`.
N4295, P0036R0	Fold expressions	A convenient syntax for applying a binary operator iteratively to the elements of a parameter pack: `template <typename ...Args> auto f(Args ...args) { return (0 + ... + args); }`
P0127R2	`template <auto>`	A non-type template parameter may now be declared with placeholder type `auto`. Examples: • `template <auto X> struct constant { static constexpr auto value = X; };` • `Delegate<&MyClass::some_function>`
P0091R3, P0433R2, P0512R0, P0620R0	Class template argument deduction	The template arguments of a class template may now be deduced from a constructor. For example, `pair p(1, 'x');` defines `p` as `pair<int, char>` (this is not an HTML error, the template arguments were omitted deliberately). The implicit deduction is complemented by a system of explicit deduction guides which allow authors to customise how the deduction happens, or forbid it.
P0292R2	Constexpr `if`	In a template specialization, the arms of the new `if constexpr (condition)` statement are only instantiated if the condition (which must be a constant expression) has the appropriate value.
P0305R1	Selection statements with initializer	The selection statements `if` and `switch` gain a new, optional initializer part: `if (auto it = m.find(key); it != m.end()) return it->second;`

Document	Summary	Examples, notes
P0170R1	Constexpr lambdas	Lambda expressions may now be constant expressions: `auto add = [](int a, int b) constexpr { return a + b; }; int arr[add(1, 2)];`
P0018R3	Lambda capture of `*this`	Before: `[self = *this]{ self.f(); }` Now: `[*this]{ f(); }`
P0386R2	Inline variables	In a header file: `inline int n = 10;` All definitions refer to the same entity. Implied for static constexpr class data members.
P0217R3, P0615R0	Structured bindings	`auto [it, ins] = m.try_emplace(key, a1, a2, a3);` Decomposes arrays, all-members-public classes, and user-defined types that follow a `get<N>` protocol like `pair` and `tuple`.
P0061R1	`__has_include`	A preprocessor operator to check whether an inclusion is possible.
P0188R1 P0189R1 P0212R1	Attribute `[[fallthrough]]` Attribute `[[nodiscard]]` Attribute `[[maybe_unused]]`	A new set of standardised attributes. The attributes formally have no required semantics, but implementations are encouraged to emit or suppress the appropriate diagnostics (warnings).
P0137R1	`launder`	A language support tool (an “optimisation barrier”) to allow libraries to reuse storage and access that storage through an old pointer, which was previously not allowed. (This is an expert tool for implementers and not expected to show up in “normal” code.)
P0298R3	A byte type	A new type `byte` is defined in `<cstddef>` (not in `<stddef.h>`, and only in namespace `std`!) which has the layout of `unsigned char`, shares the aliasing allowances of the existing char types, and has bitwise operations defined.

New library features

Document	Summary	Examples, notes
P0226R1	Mathematical special functions	The contents of the former international standard ISO/IEC 29124:2010 (mathematical special functions) are now part of C++. The functions were added only to `<cmath>`, not to `<math.h>`, and are only available in namespace `std`.
P0218R0, P0219R1, P0317R1, P0392R0, P0430R2, P0492R2	Filesystem	The contents of the Filesystems Technical Specification are now part of C++. The filesystems library allows portable interaction with directories and directory-like structures (listing directory contents, moving files, etc.). It is largely modelled on POSIX, but flexible enough to be implementable for a wide variety of systems.

Document	Summary	Examples, notes
P0024R2, P0336R1, P0394R4, P0452R1, P0467R2, P0502R0, P0518R1, P0523R1, P0574R1, P0623R0	Parallelism	The contents of the Parallelism Technical Specification are now part of C++. This adds new overloads, taking an additional *execution policy* argument, to many algorithms, as well as entirely new algorithms (see below). Three execution policies are supported, which respectively provide sequential, parallel, and vectorized execution.
P0024R2	New algorithms	The Parallelism Technical Specification adds several new algorithms to the standard library. They are motivated by their potential for efficient parallel execution, but are available in the usual simple form as well: `for_each_n`, `reduce`, `transform_reduce`, `exclusive_scan`, `inclusive_scan`, `transform_exclusive_scan`, `transform_inclusive_scan`. Note that `reduce` looks similar to the existing `accumulate`, but does not guarantee any particular order of operations.
P0220R2, P0254R2, P0403R1	New type: `string_view` (and `basic_string_view`)	The new `string_view` class is the preferred interface vocabulary type for APIs that need to view a string without wanting to take ownership or to modify it. It is constructible from char pointers, but all other classes that are string-like should offer conversions *to* `string_view`.
P0220R2, P0032R3, P0504R0	New type: `any`	The type `any` type-erases copyable objects. There are essentially three things you can do with an `any`: 1. put a value of type `T` into it. 2. Make a copy of it. 3. Ask it whether it contains a value of type `U` and get that value out, which succeeds if and only if `U` is `T`.
P0088R3, P0393R3, P0032R3, P0504R0, P0510R0	New class template: `variant`	A variant models a disjoint union (or discriminated union). A value of `variant<A, B, C>` contains *one of* an `A`, a `B`, or a `C` at any one time.
P0220R2, P0307R2, P0032R3, P0504R0	New class template: `optional`	An optional value. A `optional<T>` represents either a `T` value, or no value (which is signified by the tag type `nullopt_t`). In some respects this can be thought of as equivalent to `variant<nullopt_t, T>`, but with a purpose-built interface.
N4169	`invoke`	A facility to uniformly invoke callable entities. This allows users to write libraries with the same behaviour as the standard's magic *`INVOKE`* rule.
P0077R2, P0604R0	`is_invocable`, `is_invocable_r`, `invoke_result`	Traits to reason about invocability and invocation results.

Document	Summary	Examples, notes
P0067R5	Elementary string conversions	Functions `to_chars`, `from_chars` that produce or parse string representations of numbers. These are intended to form an efficient, low-level basis for a replacement for `printf` and iostream formatted operations. They follow idiomatic C++ algorithm style.
N3911	Alias template `void_t`	`template <class...> using void_t = void;` Surprisingly useful for metaprogramming, to simplify use of SFINAE.
N4389	Alias template `bool_constant`	`template <bool B> using bool_constant = integral_constant<bool, B>`
P0013R1	Logical operation metafunctions	Variadic metafunctions `conjunction`, `disjunction`, and `negation` for metaprogramming. These traits short-circuit in the metaprogramming sense: template specializations that are not required to determine the result are not instantiated.
P0185R1	Traits for SFINAE-friendly `swap`	New traits `is_{,nothrow_}swappable`, `is_{,nothrow_}swappable_with`.
LWG 2911	Trait `is_aggregate`	Whether a type is an aggregate. Useful for example to tell whether a generic type should be list- or non-list-initialized.
P0258R2	Trait `has_unique_object_representations`	This trait may be used to reason about whether certain value-based operations like comparison and hashing can be replaced with representation-based operations (e.g. `memcmp`).
P0007R1	`as_const`	Given an lvalue `x`, `as_const(x)` returns the const-qualified version. Does not bind to rvalues.
N4280	Non-member `size`, `data`, `empty`	The additional functions complement the existing free functions `begin`, `end` etc. to access containers and arrays in a uniform fashion. Note that unlike `begin`/`end`, the new functions are *not* customisation points for anything and are only provided for convenience.
P0025R0	`clamp`	`clamp(x, low, high)` returns either `x` if `x` is within the interval `[low, high]`, or the nearest bound otherwise.
P0295R0	`gcd` and `lcm`	Number-theoretic functions to compute the greatest common divisor and least common multiple of two integers.
N4508	Class `shared_mutex`	A reader-writer mutex, which can be locked in either shared or exclusive mode.

Document	Summary	Examples, notes
P0154R1	Interference sizes	Two new implementation-defined constants `hardware_{con,de}structive_interference_size` that effectively allow the platform to document its cache line sizes so that users can avoid false sharing (destructive interference) and improve locality (constructive interference). Two separate constants are defined to support heterogeneous architectures.
P0220R1	Tuple `apply`	Invokes a callable with arguments extracted from a given tuple.
P0209R2	Construction from tuples	A new function template `make_from_tuple` that initializes a value of type `T` from the elements of a given tuple. It is like `apply` applied to a constructor.
P0005R4, P0358R1	Universal negator `not_fn`	A call wrapper that negates its wrapped callable. This works with callables of any arity and replaces the old `not1` and `not2` wrappers.
P0220R1	Memory resources	A new set of components comprised of a *memory resource* base class for dynamically selectable memory providers, as well as three concrete implementations (`synchronized_pool_resource`, `unsynchronized_pool_resource`, `monotonic_buffer_resource`). See next item for use cases.
P0220R1, P0337R0	A polymorphic allocator	An allocator that uses a memory resource, which can be changed at runtime and is not part of the allocator type. Also contains convenience type aliases like `std::pmr::vector<T> = std::vector<T, polymorphic_allocator<T>>`.
P0220R1, P0253R1	Searcher functors	Substring searcher functors implementing the Boyer-Moore and Boyer-Moore-Horspool algorithms, and a `search` algorithm using those functors.

Modifications to existing features

Document	Summary	Examples, notes
N3928	Single-argument `static_assert`	The `static_assert` declaration no longer requires a second argument: `static_assert(N > 0);`
N4230	Nested namespace declarations	`namespace foo::bar { /* ... */ }`
N4051	Allow `typename` in template template parameters	`template <template <typename> typename Tmpl> struct X;` Previously, template template parameters were required to use the keyword `class`.

Document	Summary	Examples, notes
P0184R0	Range-based `for` takes separate begin/end types	The rewrite rule for `for (decl : expr)` now says `auto __begin = begin-expr; auto __end = end-expr;`, as opposed to `auto __begin = begin-expr, __end = end-expr;` before. This prepares the range-based `for` statement for the new Ranges (work in progress).
P0195R2	Pack expansion in *using-declaration*s	`template <typename ...Args> struct X : Args... { using Args::f...; };`
P0138R2	Construction for values of fixed enums	A variable of a fixed enumeration `E` can now be defined with `E e { 5 };` and no longer requires the the more cumbersome `E e { E(5) };`.
N4259	`uncaught_exceptions()`	The function `uncaught_exception` is deprecated, the new function `uncaught_exceptions` returns a count rather than a boolean. The previous feature was effectively unusable; N4152 explains the details.
N4266	Attributes in namespaces and enumerators	Namespaces and enumerators can now be annotated with attributes. This allows, for example, to deprecate namespaces or enumerators.
P0028R4	Attribute namespaces without repetition	This simplifies the use of attribute namespace qualifications when a namespace is used repeatedly.
N4279	Improved insertion for unique-key maps	`m.try_emplace(key, arg1, arg2, arg3)` does nothing if `key` already exists in the map, and otherwise inserts a new element constructed from the arguments. This interface guarantees that even if the arguments are bound to rvalue references, they are not moved from if the insertion does not take place.
P0084R2	Return type of `emplace`	Sequence containers whose `emplace{,_front,_back}` member function templates used to return `void` now return a reference to the newly inserted element. (Associative containers are not affected, since their insertion functions have always returned iterators to the relevant element.)
P0083R3, P0508R0	Splicing maps and sets	A new mechanism, *node handles*, has been added to the container library that allows transplanting elements between different map/set objects without touching the contained object. Moreover, this technique enables mutable access to key values of extracted nodes.
P0272R1	Non-const `string::data`	There is now a non-const overload of `basic_string::data` that returns a mutable pointer. Moreover, C++17 allows writing to the null terminator, provided that the value zero is written. This makes the string classes a bit more convenient to use with C-style interfaces.

Document	Summary	Examples, notes
P0156R0, P0156R2	A variadic version of `lock_guard` called `scoped_lock`	A new, variadic class template `scoped_lock<Args...>` that locks multiple lockable objects at once (using the same algorithm as `lock`) and releases them in the destructor. Initially it was suggested to simply change the definition of `lock_guard` to become variadic, but this was discovered to be a breaking change, and so instead we now have a new class template `scoped_lock` that is strictly superior to the old `lock_guard` and should be used instead.
P0006R0	Variable templates for traits	For every standard type trait `foo` with a single, static member constant `foo<Args...>::value`, there is now a variable template `foo_v<Args...>`.
P0152R1	`atomic::is_always_lock_free`	A new static member constant `is_always_lock_free` that documents whether the operations of a given atomic type are always lock-free. The existing non-static member function `is_lock_free` may give different answers for different values of the atomic type.
P0220R1, P0414R2	`shared_ptr` for arrays	The class template `shared_ptr` now supports C-style arrays by passing `T[]` or `T[N]` as the template argument, and the constructor from a raw pointer will install an appropriate array deleter.
P0163R0	`shared_ptr::weak_type`	The class `shared_ptr<T>` now has a member type `weak_type` which is `weak_ptr<T>`. This allows generic code to name the corresponding weak pointer type without having to destructure the shared pointer type.
P0030R1	Three-dimensional hypotenuse	The three-dimensional hypotenuse `hypot(x, y, z)` is added as an additional set of overloads to `<cmath>` (but not to `<math.h>` and only to namespace `std`).
P0040R3	Further uninitialized algorithms	Additional algorithms to create objects in uninitialized memory and to destroy objects. Separate versions for default- and value-initialization are included.
N4510	Incomplete type support for allocators	This change relaxes the requirements on allocators to have complete value types, and allows, for example, recursive structures like: `struct X { std::vector<X> data; };`
P0092R1, P0505R0	Changes to `<chrono>`	Adds floor, ceiling, division and rounding for time points; makes most member functions constexpr.
P0426R1	Constexpr for `char_traits`	All specializations required for `char_traits` now have constexpr member functions `length`, `compare`, `find` and `assign`, allowing string views to be more widely used in constant expressions.

Document	Summary	Examples, notes
N4387	Improving `pair` and `tuple`	This change makes the constructors of `pair` and `tuple` as explicit as the corresponding element type constructors.
P0435R1, P0548R1	Changes to `common_type`	Because it's not a new standard if we didn't make changes to `common_type`...

Miscellaneous

Document	Summary	Examples, notes
P0063R3	C++ refers to C11	The C++ standard now refers normatively to C11 (ISO/IEC 9899:2011) as "The C Standard". Not only does ISO require that references to other international standards refer to the latest published version, and not to a historic version, but this also gives us access to `aligned_alloc`, which is useful for the improvements to our dynamic memory management.
P0180R2	Reserved namespaces	All top-level namespaces of the form `std`*X*, where *X* is a sequence of digits, are reserved.
P0175R1	C library synopses	A purely editorial change: all headers of the "C library" part of the standard library are now presented as complete synopses in the C++ standard document, rather than as just lists of names. This makes the changes in semantics from C easier to appreciate (e.g. additional overloads, overloads on language linkage).
N4262 P0134R0 P0391R0 N4284	Term "forwarding reference" Term "default member initializer" Term "templated entity" Term "contiguous iterator"	These changes have no normative impact, but they establish official terminology for concepts that have so far only *emerged* from the language rules. Having precise and well-known terms simplifies talking about C++ and simplifies the specification.
P0346R1	Change "random number generator" to "random bit generator"	Similarly, this change has no normative impact, but clarifies the design and intended use of this aspect of the `<random>` facilities.

Unlisted papers

The following papers were moved at committee meetings, but their contents are too specific to call out as separate features: N3922, N4089, N4258, N4261, N4268, N4277, N4285, P0017R1, P0031R0, P0033R1, P0074R0, P0136R1, P0250R3, P0270R3, P0283R2, P0296R2, P0418R2, P0503R0, P0509R1, P0513R0, P0516R0, P0517R0, P0558R1, P0599R1, P0607R0, P0612R0

The following papers contain issues that have been accepted as defect reports. CWG issues are handled by N4192, N4457, P0164R0, P0167R2, P0263R1, P0384R0, P0398R0, P0490R0, P0507R0, P0519R0, P0520R0, P0522R0, P0575R1, P0576R1, P0613R0, P0622R0. LWG issues are handled by N4245, N4366,

N4383, N4525, P0165R0, P0165R1, P0165R2, P0165R2, P0165R3, P0165R4, P0304R1, P0397R0, P0610R0, P0625R0. Only specific issues may have been selected from each paper; the meeting minutes contain the details.

Assorted snippets demonstrating C++17

```
std::unordered_map<std::string, std::unique_ptr<Foo>> items;
std::vector<std::unique_ptr<Foo>> standby;

// If there is currently no item 'id', installs 'foo' as item 'id'.
// Otherwise stores 'foo' for later use and puts it on standby.

// Before C++17

void f(std::string id, std::unique_ptr<Foo> foo) {
  auto it = items.find(id);
  if (it == items.end()) {
    auto p = items.emplace(std::move(id), std::move(foo));
    p.first->second->launch();
  } else {
    standby.push_back(std::move(foo));
    standby.back()->wait_for_notification();
  }

  // Notes:
  // * Variable 'id' can no longer be used (moved-from); or...
  // * ...would need to use parameter 'const string& id' and force copying.
  // * Map lookup performed twice. Ordered map could use lower_bound + hint, but
unordered map cannot.
  // * (Cannot emplace unconditionally, because it might destroy *foo.)
}

// With C++17

void f(std::string_view id, std::unique_ptr<Foo> foo) {
  if (auto [pos, inserted] = items.try_emplace(id, std::move(foo)); inserted) {
    pos->second->launch();
  } else {
    standby.emplace_back(std::move(foo))->wait_for_notification();
  }
}
```

The next snippet illustrates the utility of `template <auto>` on the example of a class template which delegates a free function call to a member function bound to a class instance, and the member function is part of the delegate *type*.

```
// Before C++17

template <typename T, int (T::* MF)(int, int)>   // two params: one type, one non-
type
struct Delegate { /* ... */ };

int n = Delegate<MyComplexClass, &MyComplexClass::an_imporant_function>(&obj)(10,
20);
```

```
// With C++17

template <auto> struct Delegate;                // one (non-type) param
template <typename T, int (T::* MF)(int, int)>
struct Delegate<MF> { /* ... */ };              // implement as before, but as
partial specialization

int n = Delegate<&MyComplexClass::an_imporant_function>(&obj)(10, 20);
```

The next snippet shows the utility of fold expressions in generic code.

```
// Call f(n) for all f in the pack.
template <typename ...F>
void ApplyAll(int n, const F&... f) {
  (f(n), ...); // unary fold (over the comma operator)
}

// Compute f(a, b) for each f in the pack and return the sum.
template <typename ...F>
int ApplyAndSum(int a, int b, const F&... f) {
  return (f(a, b) + ... + 0); // binary fold
}
```

The next snippet shows array support for shared pointers.

```
// Before C++17
std::shared_ptr<char> p(new char[N], std::default_delete<char[]>()); // would be
wrong without the deleter

// With C++17
std::shared_ptr<char[]> p(new char[N]); // deleter uses “delete[]”
```