Mandating Annex D
Require No More

Contents
1 Abstract 1
2 Revision history 1
   2.1 R2: June 2023 (LWG review) 1
   2.2 R1: June 2023 (Varna meeting) 2
   2.3 R0: 2023 May (pre-Varna mailing) 2
3 Introduction 2
4 History 2
5 Analysis 2
6 Proposed wording 3
   6.1 Wording Intent 3
   6.2 Proposed Changes 3
7 Acknowledgements 11
8 References 11

1 Abstract
Annex D of the C++ Standard, deprecated features, retains pre-C++ 20 wording style for several library clauses. This paper updates that wording to our current standards.

2 Revision history.

2.1 R2: June 2023 (LWG review)
   — Italicised POD class and POD type when defining those terms
   — Switched from Mandates: to Preconditions: for is_pod
   — Corrected more “shall (not) be” to “is (not)”
   — Merged adjacent Preconditions: throughout
   — Added a missing “null” from the original text (that is retained)
   — Renumbered D.24 [depr.util.smartptr.shared.atomic]
   — Fixed broken mark-up when denoting semi-open ranges
2.2 R1: June 2023 (Varna meeting)
- Updated wording, based on \[N4950\]
- Removed syntax highlighting in draft wording

2.3 R0: 2023 May (pre-Varna mailing)
Initial draft of this paper, based on \[P2139R2\].
Notable changes since \[P2139R2\]
- Updated wording, based on \[N4928\]

3 Introduction
At the start of the C++23 cycle, \[P2139R2\] tried to review each deprecated feature of C++ to see which we would benefit from actively removing and which might now be better undeprecated. Consolidating all this analysis into one place was intended to ease the (L)EWG review process but in return gave the author so much feedback that the next revision of the paper was not completed.

For the C++26 cycle, a much shorter paper, \[P2863R0\], will track the overall analysis, but for features that the author wants to actively progress, a distinct paper will decouple progress from the larger paper so that the delays on a single feature do not hold up progress on all.

This paper takes up the deprecated Requires clause wording style retained by D.11 \[depr.res.on.required\].

4 History
The deprecated Requires clause wording had been replaced by the use of three new paragraph types: Constraints, Mandates, and Preconditions. The guidelines on when to use each of these new elements was proved by Walter Brown in \[P0788R3\] (which uses Expects rather than Preconditions).

A long sequence of papers by Marshall Clow provided the necessary analysis and detail to update each standard library clause. However, given priorities competing for library time, it was not deemed a good use of LWG time to complete the process for the deprecated features in Annex D, and the Editors moved the specification for Requires elements into Annex D, rather than striking it entirely, to provide consistency in the standard itself.

As features continue to be deprecated in subsequent standards, Annex D is filling up with new additions that are indeed specified in the modern style. Rather than wait indefinitely for the deprecated features with legacy wording to all be removed, this paper proposes making the final library wording updates and removing the Requires: element now.

5 Analysis
The Constraints element is used to signify a condition that should exclude an overload from name look-up rather than cause an immediate error (the SFINAE principle) and none of the wording remaining in Annex D needs this term.

The Mandates element requires the compilation produce a diagnosable error when violated by the caller. This restriction is enforced at compiler-time. Typically, when a specification has a Requires clause with type constraints, it is rewritten as a Mandates.

The Preconditions element specified run-time requirements on callers, typically on the values of provided arguments or object state.

When referring to Cpp17Requirements, use the phrase “meets the requirements”

Fix the misuse of “shall”: shall (not) be -> is (not)
6 Proposed wording

All changes are relative to [N4950].

6.1 Wording Intent

By observing whether a given Requires element applies to compile-time or run-time entities, we turn the Requires element into a Mandates or a Preconditions element.

Just the deprecated clauses that feature Requires elements are presented. A proper review of this wording should audit that there are no remaining uses elsewhere in Annex D, or the standard at large, to verify that the author’s due diligence is correct.

6.2 Proposed Changes

D.11 [depr.res.on.required] Requires paragraph

1 In addition to the elements specified in 16.3.2.4 [structure.specifications], descriptions of function semantics may also contain a Requires: element to denote the preconditions for calling a function.

2 Violation of any preconditions specified in a function’s Requires: element results in undefined behavior unless the function’s Throws: element specifies throwing an exception when the precondition is violated.

D.14 [depr.relops] Relational operators

1 The header <utility> (22.2.1 [utility.syn]) has the following additions:

```cpp
namespace std::rel_ops {
    template<class T> bool operator!=(const T&, const T&);
    template<class T> bool operator>(const T&, const T&);
    template<class T> bool operator<=(const T&, const T&);
    template<class T> bool operator>=(const T&, const T&);
}
```

2 To avoid redundant definitions of operator!= out of operator== and operators >, <=, and >= out of operator<, the library provides the following:

```cpp
template<class T> bool operator!=(const T& x, const T& y);
```

3 Requires: Type T is Cpp17EqualityComparable (Table 28).

Preconditions: T meets the Cpp17EqualityComparable requirements (Table 28).

4 Returns: !(x == y).

```cpp
template<class T> bool operator>(const T& x, const T& y);
```

5 Requires: Type T is Cpp17LessThanComparable (Table 29).

Preconditions: T meets the Cpp17LessThanComparable requirements (Table 29).

6 Returns: y < x.

```cpp
template<class T> bool operator<=(const T& x, const T& y);
```

7 Requires: Type T is Cpp17LessThanComparable (Table 29).

Preconditions: T meets the Cpp17LessThanComparable requirements (Table 29).

8 Returns: !(y < x).

```cpp
template<class T> bool operator>=(const T& x, const T& y);
```
9 **Requires:** Type T is *Cpp17LessThanComparable* (Table 29).

**Preconditions:** T meets the *Cpp17LessThanComparable* requirements (Table 29).

10 **Returns:** !(x < y).

### D.19 [depr.meta.types] Deprecated type traits

1 The header `<type_traits>` (21.3.3 [meta.type.synop]) has the following additions:

```cpp
namespace std {
    template<class T> struct is_pod;
    template<class T> constexpr bool is_pod_v = is_pod<T>::value;
    template<size_t Len, size_t Align = default_alignment> // see below
        struct aligned_storage;
    template<size_t Len, size_t Align = default_alignment> // see below
        using aligned_storage_t = typename aligned_storage<Len, Align>::type;
    template<size_t Len, class... Types>
        struct aligned_union;
    template<size_t Len, class... Types>
        using aligned_union_t = typename aligned_union<Len, Types...>::type;
}
```

2 The behavior of a program that adds specializations for any of the templates defined in this subclause is undefined, unless explicitly permitted by the specification of the corresponding template.

```cpp
template<class T> struct is_pod;
```

X A *POD class* is a class that is both a trivial class and a standard-layout class, and has no non-static data members of type non-POD class (or array thereof). A *POD type* is a scalar type, a POD class, an array of such a type, or a cv-qualified version of one of these types.

3 **Requires Preconditions:** `remove_all_extents_t<T>` shall be a complete type or cv void.

4 **Remarks:** `is_pod<T>` is a *Cpp17UnaryTypeTrait* (21.3.2 [meta.rqmts]) with a base characteristic of `true_type` if T is a POD type, and `false_type` otherwise. A POD class is a class that is both a trivial class and a standard layout class, and has no non static data members of type non POD class (or array thereof). A POD type is a scalar type, a POD class, an array of such a type, or a cv qualified version of one of these types.

5 [Note 1: It is unspecified whether a closure type (7.5.5.2 [expr.prim.lambda.closure]) is a POD type. —end note]

### D.24 [depr.util.smartptr.shared.atomic] Deprecated shared_ptr atomic access

1 The header `<memory>` (20.2.2 [memory.syn]) has the following additions:

```cpp
namespace std {
    template <class T>
        bool atomic_is_lock_free(const shared_ptr<T>* p);
    template <class T>
        shared_ptr<T> atomic_load(const shared_ptr<T>* p);
    template <class T>
        shared_ptr<T> atomic_load_explicit(const shared_ptr<T>* p, memory_order mo);
    template <class T>
        void atomic_store(shared_ptr<T>* p, shared_ptr<T> r);
    template <class T>
        void atomic_store_explicit(shared_ptr<T>* p, shared_ptr<T> r, memory_order mo);
    template <class T>
        shared_ptr<T> atomic_exchange(shared_ptr<T>* p, shared_ptr<T> r);
    template <class T>
        void atomic_exchange_explicit(shared_ptr<T>* p, shared_ptr<T> r, memory_order mo);
}```
shared_ptr<T> atomic_exchange_explicit(shared_ptr<T>* p, shared_ptr<T> r, memory_order mo);

template <class T>
bool atomic_compare_exchange_weak(
    shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w);

template <class T>
bool atomic_compare_exchange_strong(
    shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w);

template <class T>
bool atomic_compare_exchange_weak_explicit(  
    shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w,
    memory_order success, memory_order failure);

template <class T>
bool atomic_compare_exchange_strong_explicit(
    shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w,
    memory_order success, memory_order failure);
}

Concurrent access to a shared_ptr object from multiple threads does not introduce a data race if the access is done exclusively via the functions in this section and the instance is passed as their first argument.

The meaning of the arguments of type memory_order is explained in 33.5.4 [atomics.order].

template<class T>
bool atomic_is_lock_free(const shared_ptr<T>* p);

Requires Preconditions:  p shall not be null.

Returns: true if atomic access to *p is lock-free, false otherwise.

Throws: Nothing.

template<class T>
shared_ptr<T> atomic_load(const shared_ptr<T>* p);

Requires Preconditions:  p shall not be null.

Returns: atomic_load_explicit(p, memory_order::seq_cst).

Throws: Nothing.

template<class T>
shared_ptr<T> atomic_load_explicit(const shared_ptr<T>* p, memory_order mo);

Requires Preconditions:  p shall not be null.

Requires mo shall not be is neither memory_order::release nor memory_order::acq_rel.

Returns: *p.

Throws: Nothing.

template<class T>
void atomic_store(shared_ptr<T>* p, shared_ptr<T> r);

Requires Preconditions:  p shall not be null.

Effects: As if by atomic_store_explicit(p, r, memory_order::seq_cst).

Throws: Nothing.
template<class T>
    void atomic_store_explicit(shared_ptr<T>* p, shared_ptr<T> r, memory_order mo);

17 Requires Preconditions: p shall not be is not null.
18 Requires: mo shall not be is neither memory_order::acquire nor memory_order::acq_rel.
19 Effects: As if by p->swap(r).
20 Throws: Nothing.

template<class T>
    shared_ptr<T> atomic_exchange(shared_ptr<T>* p, shared_ptr<T> r);
21 Requires Preconditions: p shall not be is not null.
22 Returns: atomic_exchange_explicit(p, r, memory_order::seq_cst).
23 Throws: Nothing.

template<class T>
    shared_ptr<T> atomic_exchange_explicit(shared_ptr<T>* p, shared_ptr<T> r, memory_order mo);
24 Requires Preconditions: p shall not be is not null.
25 Effects: As if by p->swap(r).
26 Returns: The previous value of *p.
27 Throws: Nothing.

template<class T>
    bool atomic_compare_exchange_weak(shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w);
28 Requires Preconditions: p shall not be is not null.
29 Returns: atomic_compare_exchange_weak_explicit(p, v, w, memory_order::seq_cst, memory_order::seq_cst).
30 Throws: Nothing.

template<class T>
    bool atomic_compare_exchange_strong(shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w);
31 Returns: atomic_compare_exchange_strong_explicit(p, v, w, memory_order::seq_cst, memory_order::seq_cst).
template <class T>
    bool atomic_compare_exchange_weak_explicit(
        shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w, memory_order success, memory_order failure);
template <class T>
    bool atomic_compare_exchange_strong_explicit(
        shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w, memory_order success, memory_order failure);
32 Requires Preconditions: p shall not be is not null and v shall not be is not null. The failure argument shall not be is neither memory_order::release nor memory_order::acq_rel.
33 Effects: If *p is equivalent to *v, assigns w to *p and has synchronization semantics corresponding to the value of success, otherwise assigns *p to *v and has synchronization semantics corresponding to the value of failure.
34 Returns: true if *p was equivalent to *v, false otherwise.
35 Throws: Nothing.
Remarks: Two shared_ptr objects are equivalent if they store the same pointer value and share ownership. The weak form may fail spuriously. See 33.5.8.2 [atomics.types.operations].

D.27 [depr.conversions] Deprecated convenience conversion interfaces

D.27.1 [depr.conversions.general] General

The header <locale> (30.2 [locale.syn]) has the following additions:

```cpp
namespace std {
    template<class Codecvt, class Elem = wchar_t,
             class WideAlloc = allocator<Elem>,
             class ByteAlloc = allocator<char>>
    class wstring_convert;

    template<class Codecvt, class Elem = wchar_t,
             class Tr = char_traits<Elem>>
    class wbuffer_convert;
}
```

D.27.2 [depr.conversions.string] Class template wstring_convert

Class template `wstring_convert` performs conversions between a wide string and a byte string. It lets you specify a code conversion facet (like class template `codecvt`) to perform the conversions, without affecting any streams or locales.

[Example 1: If you want to use the code conversion facet `codecvt_utf8` to output to `cout` a UTF-8 multibyte sequence corresponding to a wide string, but you don’t want to alter the locale for `cout`, you can write something like:

```cpp
std::wstring_convert<std::codecvt_utf8<wchar_t>> myconv;
std::string mbstring = myconv.to_bytes(L"Hello\n");
std::cout << mbstring;
```

— end example —

```cpp
namespace std {
    template <class Codecvt, class Elem = wchar_t,
              class WideAlloc = allocator<Elem>,
              class ByteAlloc = allocator<char>>
    class wstring_convert {
        public:
            using byte_string = basic_string<char, char_traits<char>, ByteAlloc>;
            using wide_string = basic_string<Elem, char_traits<Elem>, WideAlloc>;
            using state_type = typename Codecvt::state_type;
            using int_type = typename wide_string::traits_type::int_type;

            wstring_convert() : wstring_convert(new Codecvt) {} // wstring_convert(Codecvt* pcvt = new Codecvt);
            wstring_convert(Codecvt* pcvt, state_type state);
            explicit wstring_convert(Codecvt* pcvt, state_type state);
            explicit wstring_convert(const byte_string& byte_err, const wide_string& wide_err = wide_string());
            ~wstring_convert();

            wstring_convert(const wstring_convert&) = delete;
            wstring_convert& operator=(const wstring_convert&) = delete;

            wide_string from_bytes(char byte);
            wide_string from_bytes(const char* ptr);
```
The class template describes an object that controls conversions between wide string objects of class `basic_string<Elem, char_traits<Elem>, WideAlloc>` and byte string objects of class `basic_string<char, char_traits<char>, ByteAlloc>`. The class template defines the types `wide_string` and `byte_string` as synonyms for these two types. Conversion between a sequence of `Elem` values (stored in a `wide_string` object) and multibyte sequences (stored in a `byte_string` object) is performed by an object of class `Codecvt`, which meets the requirements of the standard code-conversion facet `codecvt<Elem, char, mbstate_t>`. An object of this class template stores:

- `byte_err_string` — a byte string to display on errors
- `wide_err_string` — a wide string to display on errors
- `cvtptr` — a pointer to the allocated conversion object (which is freed when the `wstring_convert` object is destroyed)
- `cvtstate` — a conversion state object
- `cvtcount` — a conversion count

```cpp
size_t converted() const noexcept;
```

Returns: `cvtcount`.

```cpp
wide_string from_bytes(const byte_string& str);
wide_string from_bytes(const char* first, const char* last);
byte_string to_bytes(Elem wchar);
byte_string to_bytes(const Elem* wptr);
byte_string to_bytes(const wide_string& wstr);
byte_string to_bytes(const Elem* first, const Elem* last);
```

Effects: The first member function `shall` converts the single-element sequence `byte` to a wide string. The second member function `shall` converts the null-terminated sequence beginning at `ptr` to a wide string. The third member function `shall` converts the sequence stored in `str` to a wide string. The fourth member function `shall` converts the sequence defined by the range `[first, last)` to a wide string.

In all cases:

- If the `cvtstate` object was not constructed with an explicit value, it `shall` be set to its default value (the initial conversion state) before the conversion begins. Otherwise it `shall` be left unchanged.
- The number of input elements successfully converted `shall` be stored in `cvtcount`.

Returns: If no conversion error occurs, the member function `shall` returns the converted wide string. Otherwise,
if the object was constructed with a wide-error string, the member function shall return the wide-error string. Otherwise, the member function throws an object of class range_error.

state_type state() const;

8 Returns: cvtstate.

byte_string to_bytes(Elem wchar);
byte_string to_bytes(const Elem* wptr);
byte_string to_bytes(const wide_string& wstr);
byte_string to_bytes(const Elem* first, const Elem* last);

9 Effects: The first member function shall convert the single-element sequence wchar to a byte string. The second member function shall convert the null-terminated sequence beginning at wptr to a byte string. The third member function shall convert the sequence stored in wstr to a byte string. The fourth member function shall convert the sequence defined by the range \( [\text{first}, \text{last}) \) to a byte string.

In all cases:

— If the cvtstate object was not constructed with an explicit value, it shall be set to its default value (the initial conversion state) before the conversion begins. Otherwise it shall be left unchanged.

— The number of input elements successfully converted shall be stored in cvtcount.

10 Returns: If no conversion error occurs, the member function shall return the converted byte string. Otherwise, if the object was constructed with a byte-error string, the member function shall return the byte-error string. Otherwise, the member function shall throw an object of class range_error.

explicit wstring_convert(Codecvt* pcvt = new Codecvt);
wstring_convert(Codecvt* pcvt, state_type state);
explicit wstring_convert(const byte_string& byte_err,
const wide_string& wide_err = wide_string());

12 Requires Preconditions: For the first and second constructors, pcvt ≠ nullptr is not null.

13 Effects: The first constructor shall store pcvt in cvtpt and default values in cvtstate, byte_err_string, and wide_err_string. The second constructor shall store pcvt in cvtpt, state in cvtstate, and default values in byte_err_string and wide_err_string; moreover the stored state shall be retained between calls to from_bytes and to_bytes. The third constructor shall stores new Codecvt in cvtpt, state_type() in cvtstate, byte_err in byte_err_string, and wide_err in wide_err_string.

~wstring_convert();

14 Effects: The destructor shall delete delete cvtpt.

D.27.3 [depr.conversions.buffer] Class template wbuffer_convert

1 Class template wbuffer_convert looks like a wide stream buffer, but performs all its I/O through an underlying byte stream buffer that you specify when you construct it. Like class template wstring_convert, it lets you specify a code conversion facet to perform the conversions, without affecting any streams or locales.

namespace std {
  template <class Codecvt, class Elem = wchar_t, class Tr = char_traits<Elem>>
  class wbuffer_convert
    : public basic_streambuf<Elem, Tr> {
    public:
      using state_type = typename Codecvt::state_type;

      wbuffer_convert() : wbuffer_convert(nullptr) {}
      explicit wbuffer_convert(streambuf* bytebuf = 0,
                               Codecvt* pcvt = new Codecvt,
state_type state = state_type();

~wbuffer_convert();

wbuffer_convert(const wbuffer_convert&) = delete;
wbuffer_convert& operator=(const wbuffer_convert&) = delete;

streambuf* rdbuf() const;
streambuf* rdbuf(streambuf* bytebuf);

state_type state() const;

private:
    streambuf* bufptr; // exposition only
    Codecvt* cvtptr; // exposition only
    state_type cvtstate; // exposition only
};

2 The class template describes a stream buffer that controls the transmission of elements of type Elem, whose character traits are described by the class Tr, to and from a byte stream buffer of type streambuf. Conversion between a sequence of Elem values and multibyte sequences is performed by an object of class Codecvt, which shall meet the requirements of the standard code-conversion facet codecvt<Elem, char, mbstate_t>.

3 An object of this class template stores:
   — bufptr — a pointer to its underlying byte stream buffer
   — cvtptr — a pointer to the allocated conversion object (which is freed when the wbuffer_convert object is destroyed)
   — cvtstate — a conversion state object

state_type state() const;

4 Returns: cvtstate.

streambuf* rdbuf() const;

5 Returns: bufptr.

streambuf* rdbuf(streambuf* bytebuf);

6 Effects: Stores bytebuf in bufptr.

7 Returns: The previous value of bufptr.

explicit wbuffer_convert(streambuf* bytebuf = 0,
    Codecvt* pcvt = new Codecvt, state_type state = state_type());

8 Requires Preconditions: pcvt != nullptr is not null.

9 Effects: The constructor constructs a stream buffer object, initializes bufptr to bytebuf, initializes cvtptr to pcvt, and initializes cvtstate to state.

~wbuffer_convert();

10 Effects: The destructor shall delete cvtptr.

D.29 [depr.fs.path.factory] Deprecated filesystem path factory functions

The header <filesystem> (31.12.4 [fs.filesystem.syn]) has the following additions:
template<class Source>
  path u8path(const Source& source);

template<class InputIterator>
  path u8path(InputIterator first, InputIterator last);

Requires: The `source` and `[first, last)` sequences are UTF-8 encoded. The value type of `Source` and `InputIterator` is `char` or `char8_t`. Source meets the requirements specified in 31.12.6.4 [fs.path.req].

Mandates: The value type of `Source` and `InputIterator` is `char` or `char8_t`.

Preconditions: The `source` and `[first, last)` sequences are UTF-8 encoded. `Source` meets the requirements specified in 31.12.6.4 [fs.path.req].

Returns:

- If `path::value_type` is `char` and the current native narrow encoding (31.12.6.3.2 [fs.path.type.cvt]) is UTF-8, return `path(source)` or `path(first, last)`; otherwise,
- if `path::value_type` is `wchar_t` and the native wide encoding is UTF-16, or if `path::value_type` is `char16_t` or `char32_t`, convert `source` or `[first, last)` to a temporary, `tmp`, of type `path::string_type` and return `path(tmp)`; otherwise,
- convert `source` or `[first, last)` to a temporary, `tmp`, of type `u32string` and return `path(tmp)`.

Remarks: Argument format conversion (31.12.6.3.1 [fs.path.fmt.cvt]) applies to the arguments for these functions. How Unicode encoding conversions are performed is unspecified.

Example 1: 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:

```cpp
namespace fs = std::filesystem;
std::string utf8_string = read_utf8_data();
fs::create_directory(fs::u8path(utf8_string));
```

For POSIX-based operating systems with the native narrow encoding set to UTF-8, no encoding or type conversion occurs.

For POSIX-based operating systems with the native narrow encoding not set to UTF-8, a conversion to UTF-32 occurs, followed by a conversion to the current native narrow encoding. Some Unicode characters may have no native character set representation.

For Windows-based operating systems a conversion from UTF-8 to UTF-16 occurs. —end example]

Note 1: The example above is representative of a historical use of `filesystem::u8path`. To indicate a UTF-8 encoding, passing a `std::u8string` to `path`’s constructor is preferred as it is consistent with `path`’s handling of other encodings. —end note]

7 Acknowledgements

Thanks to Michael Park for the pandoc-based framework used to transform this document’s source from Markdown.

8 References


