

# P1030R8: `std::filesystem::path_view`

Document #: P1030R8  
Date: 2024-12-12  
Project: Programming Language C++  
Library Evolution Working Group  
Reply-to: Niall Douglas  
<[s\\_sourceforge@nedprod.com](mailto:s_sourceforge@nedprod.com)>

A proposed wording for a `std::filesystem::path_view_component` and `std::filesystem::path_view`, a non-owning view of explicitly unencoded or encoded character sequences in the format of a native or generic filesystem path, or a view of a binary key. In the Prague 2020 meeting, LEWG requested IS wording for this proposal targeting the C++ 23 standard release.

There are lengthy, ‘persuasive’, arguments about design rationale in R3 (<https://wg21.link/P1030R3>). From R4 onwards, this has been condensed into a set of design goals and change tracking log.

If you wish to use an implementation right now, a highly-conforming reference implementation of the proposed path view can be found at [https://github.com/ned14/llfio/blob/master/include/llfio/v2.0/path\\_view.hpp](https://github.com/ned14/llfio/blob/master/include/llfio/v2.0/path_view.hpp). It has been found to work well on recent editions of GCC, clang and Microsoft Visual Studio, on x86, x64, ARM and AArch64. It has been in production use for several years now.

## Contents

<b>1</b>	<b>Design goals</b>	<b>2</b>
1.1	<code>path_view_component</code> and <code>path_view</code> . . . . .	2
1.2	<code>path_view::rendered_path</code> . . . . .	4
<b>2</b>	<b>ABI proofs</b>	<b>5</b>
2.1	Free function path taking overloads . . . . .	5
2.2	Adding a new encoding in the future . . . . .	7
<b>3</b>	<b>Change tracking log for LWG since R4</b>	<b>9</b>
<b>4</b>	<b>Example of ‘full fat’ use as per LEWG request</b>	<b>12</b>
<b>5</b>	<b>Delta from N4986</b>	<b>13</b>
<b>6</b>	<b>Acknowledgements</b>	<b>53</b>
<b>7</b>	<b>References</b>	<b>53</b>

# 1 Design goals

## 1.1 `path_view_component` and `path_view`

- Path and path component views implement a non-owning, trivially copyable, runtime variant view instead of a compile time typed view such as `basic_string_view<CharT>`. They can represent backing data in one of:
  - The narrow system encoding (`char`).
  - The wide system encoding (`wchar_t`).
  - UTF-8 encoding (`char8_t`).
  - UTF-16 encoding (`char16_t`).
  - Unencoded raw bytes (`byte`).

LEWG has decided that `char32_t` is explicitly omitted for now, it could be added in a future standard if needed.

- Path views, like paths, have an associated *format*, which reuses and extends `filesystem::format`:
  - `format::native_format`: The path's components are to be separated if needed by C++ only using the native separator only. Platform APIs may parse separation independently.
  - `format::generic_format`: The path's components are to be separated if needed by C++ only using the generic separator only. Platform APIs may parse separation independently.
  - `format::auto_format`: The path's components are to be separated if needed by C++ only using *either* the native or generic separators (and in the case of path views, any mix thereof). Platform APIs may parse separation independently.
  - `format::binary_format`: The path's components are not to be separated if needed by C++ only in any way at all. Platform APIs may parse separation independently.
- When a path view is iterated, it yields a path view component as according to the formatting set for that path view. A path view component cannot be iterated, as it is considered to represent a path which is not separated by path separators, however it still carries knowledge of its formatting as that may be used during rendition of the view to other formats.

Constructing a path view component directly defaults to `format::binary_format` i.e. do not have C++ treat path separators as separators (this applies to the standard library only, not to platform APIs). It is intentionally possible to construct a path view component directly with other formatting, as an example this might induce the conversion of generic path separators to native path separators in path view consumers.

Path views, like paths, have default formatting of `format::auto_format`.

- Whilst the principle use case is expected to target file systems whose native filesystem encoding is `filesystem::path::value_type`, the design is generic to all kinds of path usage e.g. within a

ZIP archiver library where paths may be hard coded to the narrow system encoding, or within Java JNI where paths are hard coded to UTF-16 on all platforms.

- The design is intended to be Freestanding C++ compatible, albeit that if dynamic memory allocation or reencoding were required, neither would ever succeed unless a custom allocator were supplied. Thus path views ought to be available and usable without `path` being available, which is why it has been placed in its own header `<path_view>`. The design has an obvious implementation defined behaviour if exceptions are globally disabled.

(This is to make possible a read-only ‘fake filesystem’ embeddable into the program binary which could help improve the portability of hosted C++ code to freestanding)

- Path views provide identity-based comparisons rather than across-encodings-based. There is a separate, potentially relatively very high cost, contents-after-reencode-comparing comparison function. Comparisons where path views may implicitly construct from literals are deleted to avoid end user performance surprises.
- Path view consuming APIs determine how path views ought to be interpreted on a case by case basis, and this is generally implementation defined. For example, if the view consuming API is wrapping a file system, and that file system might support binary key file content lookup, the view consuming API may interpret unencoded raw byte input as a binary key, returning a failure if the target file system does not when questioned at runtime support binary keys.

A path view consumer may reject unencoded raw byte input by throwing an exception or other mode of failure – indeed `filesystem::path` is exactly one such consumer.

- A number of convenience renderers of path views to a destination format are provided:
  - `filesystem::path`’s constructors can accept all backing data encodings except unencoded raw bytes<sup>1</sup>, and we provide convenience path view accepting constructor overloads which `visit()` the backing data and construct a path from that. These additional constructors on `path` are `explicit` to prevent hidden performance impact surprises.
  - `path_view.render_null_terminated()` and `path_view.render_unterminated()` will render a path view to a destination encoding and null termination using an internal buffer to avoid dynamic memory allocation. See detail below.
- `path_view` inherits publicly from `path_view_component`, and contains no additional member data. `path_view` can be implicitly constructed from `path_view_component`. Thus both types are implicitly convertible from and into one another. Note however that the formatting setting is propagated unchanged during conversion, which whilst not ideal, is considered to be the least worst of the choices available.
- Finally, two extra free function overloads are added for `path` which fix performance issues and make path more consistent with the rest of the standard library.

---

<sup>1</sup>It would be preferable if paths could also represent unencoded raw bytes, but they would need a completely different design, and it could not be binary compatible with existing path implementations.

## 1.2 `path_view::rendered_path`

- `render_null_terminated()` and `render_unterminated()` returning a `path_view::rendered_path` is expected to be the most commonly used mechanism in newly written code for rendering a path view ready for consumption by a platform syscall, or C function accepting a zero terminated codepoint array. If the user supplies backing data in a compatible encoding to the destination encoding, reencoding can be avoided. If the user supplies backing data which is zero terminated, or the destination does not require zero termination according to the parameters supplied to `render_*()`, memory copying can be avoided. For the vast majority of C++ code on POSIX platforms when targeting the filesystem, reencoding is always avoided and memory copying is usually avoided due to C++ source code string literals having a compatible encoding with filesystem paths.
- For the default configuration of `rendered_path`, dynamic memory allocation is usually avoided through the use of a reasonably large inline buffer. This makes `rendered_path` markedly larger than most classes typically standardised by the committee (expected to be between 1Kb and 2Kb depending on platform, but actual size is chosen by implementers). The intent is that `rendered_path` will be instantiated on the stack immediately preceding a syscall to render the path view into an appropriate form for that syscall. Upon the syscall's return, the `rendered_path` is unwound in the usual way. Therefore the large size is not the problem it might otherwise be.
- `rendered_path` is intended to be storable within STL containers as that can be useful sometimes, and provides assignment so a single stack allocated `rendered_path` instance can be reused for multiple path view inputs during a function. Via template parameters, `rendered_path` can be forced to be small for any particular use case, and thus exclusively use dynamic memory allocation. Similarly, via template parameters one can force `rendered_path` to be as large as the maximum possible path (e.g. `PATH_MAX`) and thus guarantee that no dynamic memory allocation can ever occur.
- For typical end users, `rendered_path` is expected to almost always be used with its default dynamic memory allocator, which uses an implementation defined allocator (this permits avoidance of an unnecessary extra dynamic memory allocation and memory copy on some platforms).

If one wishes to customise dynamic memory allocation, one can supply an `allocator` instance as a parameter:

```
1 namespace detail {
2     struct thread_local_scratch_allocator_t {
3         char *allocate(size_t);
4         void deallocate(void *, size_t);
5     };
6 }
7
8 std::filesystem::path_view v("foo");
9 auto rpath = v.render_null_terminated(detail::thread_local_scratch_allocator_t{});
10 int fd = ::open(rpath.c_str(), O_RDONLY);
```

## 2 ABI proofs

For R7 LEWG requested proofs that backwards and forwards ABI compatibility would be unaffected.

### 2.1 Free function path taking overloads

Consider this current filesystem free function overload set for `filesystem::copy()`:

```
1 void copy(const path& from, const path& to);
2 void copy(const path& from, const path& to, error_code& ec);
```

This paper would extend that overload set to:

```
1 void copy(const path& from, const path& to);
2 void copy(const path& from, const path& to, error_code& ec);
3 void copy(path-view-like from, path-view-like to);
4 void copy(path-view-like from, path-view-like to, error_code& ec) noexcept;
```

`path-view-like` which was introduced in R6 is an exposition-only type implicitly constructible from any type `T` for which:

- `std::is_convertible_v<T, path_view>` is `true`, and
- `std::is_convertible_v<T, path>` is `false`

R6 does not specify how to implement this as R5 did using named requirements, so as per LEWG request here is a proof that existing C++ code will continue to call the overload it originally did and that existing C++ code would need to ‘opt in’ to using the path view overloads.

Here is the `path` constructor overload set:

```
1 path() noexcept;
2
3 path(const path& p);
4 path(path&& p) noexcept;
5
6 path(string_type&& source, format fmt = auto_format);
7
8 template<class Source>
9 path(const Source& source, format fmt = auto_format);
10
11 template<class InputIterator>
12 path(InputIterator first, InputIterator last, format fmt = auto_format);
13
14 template<class Source>
15 path(const Source& source, const locale& loc, format fmt = auto_format);
16
17 template<class InputIterator>
18 path(InputIterator first, InputIterator last, const locale& loc, format fmt = auto_format);
19
20 // Added by this proposal
21 explicit path(path-view-like p);
```

Here is the proposed `path_view` constructor set:

```
1  constexpr path_view() noexcept;
2
3  path_view(path_view_component p, format fmt = path::auto_format) noexcept;
4
5  path_view(const path& p) noexcept;
6  template<class CharT>
7  constexpr path_view(const basic_string<CharT>,
8                      format fmt = path::auto_format) noexcept;
9
10 template<class CharT>
11 constexpr path_view(const CharT* b, size_type l, enum termination zt,
12                    format fmt = path::auto_format) noexcept;
13 constexpr path_view(const byte* b, size_type l, enum termination zt) noexcept;
14
15 template<class CharT>
16 constexpr path_view(const CharT* b, format fmt = path::auto_format) noexcept;
17 constexpr path_view(const byte* b) noexcept;
18
19 template<class CharT>
20 constexpr path_view(basic_string_view<CharT> b, enum termination zt,
21                    format fmt = path::auto_format) noexcept;
22 constexpr path_view(span<const byte> b, enum termination zt) noexcept;
23
24 template<class It, class End>
25 constexpr path_view(It b, End e, enum termination zt,
26                    format fmt = path::auto_format) noexcept;
27 template<class It, class End>
28 constexpr path_view(It b, End e, enum termination zt) noexcept;
29
30 constexpr path_view(const path_view&) = default;
31 constexpr path_view(path_view&&) = default;
```

`path-view-like` would therefore accept:

```
1  path_view(path_view_component p, format fmt = path::auto_format) noexcept;
2
3  template<class CharT>
4  constexpr path_view(const CharT* b, size_type l, enum termination zt,
5                      format fmt = path::auto_format) noexcept;
6  constexpr path_view(const byte* b, size_type l, enum termination zt) noexcept;
7
8  template<class CharT>
9  constexpr path_view(basic_string_view<CharT> b, enum termination zt,
10                    format fmt = path::auto_format) noexcept;
11 constexpr path_view(span<const byte> b, enum termination zt) noexcept;
12
13 template<class It, class End>
14 constexpr path_view(It b, End e, enum termination zt,
15                    format fmt = path::auto_format) noexcept;
16 template<class It, class End>
17 constexpr path_view(It b, End e, enum termination zt) noexcept;
18
19 constexpr path_view(const path_view&) = default;
20 constexpr path_view(path_view&&) = default;
```

Therefore, to opt into the path view overload of the filesystem free function one has these choices:

1. Supply a `path_view` or `path_view_component`.
2. Supply an additional parameter of type `enum termination` to describe how the source of the view is terminated.
3. Supply a source of `bytes`.

As no existing valid standard C++ code can do any of these three options, therefore all existing valid standard C++ code will continue to call the existing filesystem free function overloads.

## 2.2 Adding a new encoding in the future

Some in the St. Louis meeting of LEWG expressed concern that new encodings could not be added to path views in the future without breaking ABI.

The reference implementation which is described expositionally in the wording below as follows:

```
1 private:
2     union
3     {
4         const byte* bytestr_{nullptr}; // exposition only
5         const char* charstr_; // exposition only
6         const wchar_t* wcharstr_; // exposition only
7         const char8_t* char8str_; // exposition only
8         const char16_t* char16str_; // exposition only
9     };
10    size_type length_{0}; // exposition only
11    uint16_t null_terminated_ : 1; // exposition only
12    uint16_t is_bytestr_ : 1; // exposition only
13    uint16_t is_charstr_ : 1; // exposition only
14    uint16_t is_wcharstr_ : 1; // exposition only
15    uint16_t is_char8str_ : 1; // exposition only
16    uint16_t is_char16str_ : 1; // exposition only
17    format format_{format::unknown}; // exposition only
```

If implementations chose to, this can be packed into two registers on a 64 bit architecture as follows:

```
1 private:
2     union
3     {
4         const byte* bytestr_{nullptr};
5         const char* charstr_;
6         const wchar_t* wcharstr_;
7         const char8_t* char8str_;
8         const char16_t* char16str_;
9     };
10    uint64_t length_ : 48;
11    uint64_t null_terminated_ : 1;
12    uint64_t is_bytestr_ : 1;
13    uint64_t is_charstr_ : 1;
14    uint64_t is_wcharstr_ : 1;
15    uint64_t is_char8str_ : 1;
```

```

16  uint64_t is_char16str_ : 1;
17  uint64_t reserved_ : 7;
18  uint64_t format_ : 3;

```

As you can see, this leaves seven bits free. Let us add a `char32_t` encoding:

```

1  private:
2  union
3  {
4  const byte* bytestr_{nullptr};
5  const char* charstr_;
6  const wchar_t* wcharstr_;
7  const char8_t* char8str_;
8  const char16_t* char16str_;
9  const char32_t* char32str_;
10 };
11 uint64_t length_ : 48;
12 uint64_t null_terminated_ : 1;
13 uint64_t is_bytestr_ : 1;
14 uint64_t is_charstr_ : 1;
15 uint64_t is_wcharstr_ : 1;
16 uint64_t is_char8str_ : 1;
17 uint64_t is_char16str_ : 1;
18 uint64_t is_char32str_ : 1;
19 uint64_t reserved_ : 6;
20 uint64_t format_ : 3;

```

ABI is unaffected, and up to seven new encodings could be added in the future on 64 bit. Repeating the same for 32 bit:

```

1  private:
2  union
3  {
4  const byte* bytestr_{nullptr};
5  const char* charstr_;
6  const wchar_t* wcharstr_;
7  const char8_t* char8str_;
8  const char16_t* char16str_;
9  };
10 uint32_t length_ : 24;
11 uint32_t null_terminated_ : 1;
12 uint32_t strtype_ : 3;
13 uint32_t reserved_ : 1;
14 uint32_t format_ : 3;

```

This fits into two registers on a 32 bit architecture at the cost of a maximum path view length of  $16.7M^2$  and allows for three additional encodings to be added later without breaking ABI.

(The reason that individual bits were originally used to encode type instead of a count was because on the compilers at the time this code was written, code elision by optimisers was found to work

---

<sup>2</sup>The longest kernel filesystem path possible in the major operating systems is 65,535. However path views are very useful elsewhere e.g. for environment variables, and there the limit is 2M.



best with a dedicated bit per encoding type. More recent compilers may be better and so space can be saved)

Class `rendered_path` is not expected to be used in ABI boundaries, and therefore will not have ABI compatibility concerns like `path_view`. It seems unlikely it would be significantly extended in the future in any case as its template arguments provide a customisation point.

If additional encodings were added in the future, `visit()` may break as the type supplied would be novel. It would depend on how the visitor function were implemented. The chances are high it would either be a compile-time error, or it would work correctly because the writer had designed it to be so (after all, new encodings added here are likely not going to be crazy).

### 3 Change tracking log for LWG since R4

The WG21 tracker for this paper can be found at <https://github.com/cplusplus/papers/issues/406>.

- R7 => R8:
  - Added to *path-format-spec* the ?? option which formats the path as an escaped string only if the backing representation of the view is `byte`.
- R6 => R7:
  - Rebased onto latest working draft <https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2024/n4986.pdf>.
  - For some reason the path view `render_*()` member functions were not `const`-qualified neither in this paper nor in the reference implementation, which was an oversight and has been remedied. My thanks to whomever spotted in this the SG9 Ranges meeting on `path_view` in the Kona 2023 meeting.
  - As per LEWG request, `c_str()` now only requires zero termination and no other preconditions.
  - As per LEWG request, remove all mention of `[[nodiscard]]`.
  - LEWG seemed last meeting to have finally decided that they want this to go into its own header rather than be part of `<filesystem>`. I was asked to choose a new header name. I was not innovative, and chose `<path_view>`. I have tried my best to change everywhere in the standard where I think `<path_view>` would now need to be mentioned.
  - As per LEWG request, added proof that `path-view-like` will not cause existing C++ code to call new filesystem free function overloads without modifying the source code.
  - As per LEWG request, added proof that `path_view` can have new underlying encodings added in future without breaking ABI.
  - As per LEWG request, added a formatter replicating the formatter for filesystem path.

- As per LEWG request, added normative wording explaining why many path view taking free function overloads with error code are noexcept.
  - For several meetings now, LEWG has found the example of use lacking, and expressed desires for a ‘real world’ example of use. There was not consensus on what form that should take, so I chose the ‘fullest fat’ possible example which is now given its own section due to its length. The short example in the normative wording has been removed.
  - Somebody noted that comparison operators were missing for `path_view`, fixed.
  - LEWG requested that the spaceship operator interactions with deleted comparison operators be tested in the reference implementation. This has now been done, however the reference implementation uses a static assert failure message rather than a deleted failure message as it is a C++ 14 codebase. To test that actual operator delete would work equivalently, this godbolt demonstrates it working as expected: <https://godbolt.org/z/Gqz1Wf3s4>.
  - LEWG requested fuller fat normative wording for the filesystem path taking i/o stream functions than just ‘as if’, this has been done.
- R5 => R6:
    - Remove all overloads taking a `std::locale` as per LEWG guidance.
    - Fix incorrect ostream formatter as reported by Victor (thanks!).
    - Removed `render()` as per LEWG request.
    - Removed locale-based overloads as per LEWG request.
    - Use *Returns* instead of *Effects: As if* in overloads in [filebuf].
    - Fix incorrect mentions of a free function `render_zero_terminated()` to be member functions, which was a mistake.
    - Clarify the lifetime semantics of `rendered_path` as per LEWG request.
    - LEWG requested a table of which `filesystem::path` implementations store the `format` with which they were created:
      - \* libstdc++: Ignores the format supplied in the constructor completely (throws the values away and does not store it).
      - \* MSVC: Ignores the format supplied in the constructor completely (throws the values away and does not store it).
      - \* libc++: Ignores the format supplied in the constructor completely (throws the values away and does not store it).
      - \* Boost.Filesystem: On POSIX ignores the format value; On Windows, if `native_format` requested, does not perform conversion of string before storage; if `generic_format` requested, all backslashes are converted to forward slashes in the string before storage.

The chosen solution is that if an implementation ignores the format value during construction, `filesystem::path::format()` always returns `auto_format` as that is effectively the hard coded formatting choice.

- Use *Returns* instead of *Effects: As if* in overloads in [filebuf]
  - Removed named type requirements and respecified `path_view`-accepting overloads in terms of `path-view-like` (based on techniques used in P1928R4)
  - Removed all notes for LEWG which weren't notes for the final text as per LEWG request.
  - Updated feature test macro to `YYYYMML`.
  - Removed erroneous parameter from `render_null_terminated` and `render_unterminated`.
  - Added normative wording for `render_null_terminated` and `render_unterminated`.
  - Replace 'implementation defined' with 'see later normative wording' where appropriate, as per LEWG request.
  - Changed references to "zero terminated" to reference "null terminated"
  - Changed references to "not zero termination" to reference "unterminated"
  - Apply `noexcept` in [fs.filesystem.syn] consistently with existing overloads already taking a `const path&`
  - Specify semantics of `path(path_view)` and `path_view::operator<<`
  - Add `std::hash` for `path_view_fragment`
- R4 => R5:
    - Rebased proposed wording changes to the latest IS working draft.
    - Bumped `__cpp_lib_filesystem` value to 202209L.
    - `struct c_str` was renamed to `class rendered_path`, with added convenience functions of `zero_terminated_rendered_path` and `not_zero_terminated_rendered_path`.
    - Member variables in `rendered_path` became member function accessors.
    - A new member function `rendered_path::c_str()` becomes available if and only if a zero terminated rendition is requested. This required moving the zero termination specifier into a template parameter. `rendered_path::data()` and `rendered_path::size()` are available in all use cases.
    - All custom dynamic allocation mechanisms for `rendered_path` other than STL allocators were removed, though the default 'implementation defined' dynamic allocator is retained to improve optimisation opportunities on some platforms.
    - Added `path_view::render()` as per LEWG request, with added convenience functions `path_view::render_zero_terminated()` and `path_view::render_not_zero_terminated()`.
    - Added `hash_value()` overloads for `path_view_component`, `path_view` and `rendered_path`.

## 4 Example of ‘full fat’ use as per LEWG request

I hard coded the various flags and options are they aren’t relevant.

This example handles all supported backing data forms: `char`, `wchar_t`, `char8_t`, `char16_t`, and `byte`. It also renders path views to zero terminated and unterminated forms.

It should be stressed that this isn’t a recommended implementation, just an example of potential use as the committee requested something meatier than what previous revisions had.

```
1 HANDLE opentat(HANDLE base, path_view path)
2 {
3     static constexpr DWORD access = GENERIC_READ;
4     static constexpr DWORD share =
5     FILE_SHARE_READ | FILE_SHARE_WRITE | FILE_SHARE_DELETE;
6     static constexpr DWORD creation = OPEN_EXISTING;
7     static constexpr DWORD flags = FILE_ATTRIBUTE_NORMAL;
8     return visit(
9     [&](auto sv) -> HANDLE
10    {
11        using type = typename decltype(sv)::value_type;
12        if constexpr(is_same_v<type, byte>)
13        {
14            if(sv.size() == 16)
15            {
16                FILE_ID_DESCRIPTOR fid{};
17                fid.dwSize = sizeof(fid);
18                fid.Type = 1;
19                memcpy(&fid.ObjectId, sv.data(), 16);
20                return OpenFileById(base, &fid, access, share, nullptr, flags);
21            }
22            throw std::runtime_error("binary key must be exactly 16 bytes long");
23        }
24        // A "\\!\\" or "\\?\" prefix enables direct use of the NT kernel API
25        // or setting base, as the Win32 API doesn't support paths relative to a
26        // base
27        const bool is_ntpath =
28        sv.size() >= 4 && sv[0] == '\\\' && sv[3] == '\\\' &&
29        ((sv[1] == '!\' && sv[2] == '!') || (sv[1] == '?\' && sv[2] == '?'));
30        if(base != nullptr || is_ntpath)
31        {
32            // The NT kernel always takes the system wide encoding
33            auto zpath = path.render_unterminated<wchar_t>(path);
34            UNICODE_STRING _path{};
35            _path.Buffer = const_cast<wchar_t *>(zpath.data());
36            _path.MaximumLength =
37            (_path.Length = static_cast<USHORT>(zpath.size() * sizeof(wchar_t))) +
38            sizeof(wchar_t);
39            // The "\\!\\" prefix is library local and the NT kernel doesn't
40            // understand it unlike "\\?\", so strip it off
41            if(zpath.size() >= 4 && _path.Buffer[0] == '\\\' &&
42            _path.Buffer[1] == '!\' && _path.Buffer[2] == '!\' &&
43            _path.Buffer[3] == '\\\'
44            {
45                _path.Buffer += 3;
```

```

46     _path.Length -= 3 * sizeof(wchar_t);
47     _path.MaximumLength -= 3 * sizeof(wchar_t);
48 }
49
50 OBJECT_ATTRIBUTES oa{};
51 oa.Length = sizeof(OBJECT_ATTRIBUTES);
52 oa.ObjectName = &_amp;_path;
53 oa.RootDirectory = base;
54 oa.Attributes = 0;
55
56 IO_STATUS_BLOCK isb{};
57 LARGE_INTEGER AllocationSize{};
58 HANDLE ret = INVALID_HANDLE_VALUE;
59 NTSTATUS ntstat =
60 NtCreateFile(&ret, access, &oa, &isb, &AllocationSize, 0, share,
61             FILE_OPEN, FILE_SYNCHRONOUS_IO_NONALERT, nullptr, 0);
62 if(ntstat < 0)
63 {
64     // Might want to call RtlNtStatusToDosError(ntstat)?
65 }
66 return ret;
67 }
68 if constexpr(is_same_v<type, char>)
69 {
70     // Render to the system narrow encoding null terminated
71     auto zpath = path.render_null_terminated<char>(path);
72     return CreateFileA(zpath.c_str(), access, share, nullptr, creation,
73                       flags, nullptr);
74 }
75 else // char8_t, char16_t, wchar_t
76 {
77     // Render to the system wide encoding null terminated
78     auto zpath = path.render_null_terminated<wchar_t>(path);
79     return CreateFileW(zpath.c_str(), access, share, nullptr, creation,
80                       flags, nullptr);
81 }
82 },
83 path);
84 }

```

## 5 Delta from N4986

The following normative wording delta is against <https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2024/n4986.pdf>. Green text is wording to be added, red text is wording to be removed, black text is generally notes to LEWG which shall be removed if the paper is sent to LWG.

### In 16.4.2.3 [headers] paragraph 2:

+ <path\_view>

## In 17.3.2 [version.syn] paragraph 2:

```
#define __cpp_lib_char8_t 201907L // also in <atomic>, <filesystem>, <istream>, <limits>, <locale>, <ostream>, <path_view>, <string>, <string_view>

#define __cpp_lib_filesystem 201703L // also in <filesystem>, <path_view>

#define __cpp_lib_filesystem_path_view YYYYMMML // also in <filesystem>, <path_view>
```

## In 31.1 [input.output.general] paragraph 2:

31.12 File systems <filesystem>, <path\_view>

### In 31.10.2 [filebuf]:

```
1  template<class charT, class traits = char_traits<charT>>
2  class basic_filebuf : public basic_streambuf<charT, traits> {
3      public:
4
5      // ...
6
7      // [filebuf.members], members
8      bool is_open() const;
9      basic_filebuf* open(const char* s, ios_base::openmode mode);
10     basic_filebuf* open(const filesystem::path::value_type* s,
11                         ios_base::openmode mode); // wide systems only; see [fstream.syn]
12     basic_filebuf* open(const string& s,
13                         ios_base::openmode mode);
14     basic_filebuf* open(const filesystem::path& s,
15                         ios_base::openmode mode);
```

```
+ basic_filebuf* open(path-view-like s, ios_base::openmode mode);
```

```
1  // ...
2  }
```

### In 31.10.3.4 [filebuf.members] paragraph 2:

```
basic_filebuf* open(const char* s, ios_base::openmode mode);
```

```
basic_filebuf* open(const filesystem::path::value_type* s, ios_base::openmode mode); //wide systems only
```

```
+ basic_filebuf* open(path-view-like pv, ios_base::openmode mode);
```

*Expects:* `s` points to a NTCTS.

*Effects:* If `is_open() != false`, returns a null pointer. Otherwise, initializes the `filebuf` as required. It then opens the file to which `s` or `pv` resolves, if possible, as if by a call to `fopen` with the second

argument determined from `mode` & `~ios_base::ate` as indicated in Table 131. If mode is not some combination of flags shown in the table then the open fails.

### In 31.10.4 [ifstream]:

```
1  template<class charT, class traits = char_traits<charT>>
2  class basic_ifstream : public basic_streambuf<charT, traits> {
3      public:
4
5      // ...
6
7      // [ifstream.cons], constructors
8      basic_ifstream();
9      explicit basic_ifstream(const char* s,
10                             ios_base::openmode mode = ios_base::in);
11     explicit basic_ifstream(const filesystem::path::value_type* s,
12                             ios_base::openmode mode = ios_base::in); // wide systems only; see [fstream.
13     explicit basic_ifstream(const string& s,
14                             ios_base::openmode mode = ios_base::in);
```

+ `explicit basic_ifstream(path-view-like s, ios_base::openmode mode = ios_base::in);`

```
1  template<class T>
2      explicit basic_ifstream(const T& s, ios_base::openmode mode = ios_base::in);
3  basic_ifstream(const basic_ifstream&) = delete;
4  basic_ifstream(basic_ifstream&& rhs);
5
6  basic_ifstream& operator=(const basic_ifstream&) = delete;
7  basic_ifstream& operator=(basic_ifstream&& rhs);
8
9  // [ifstream.swap], swap
10 void swap(basic_ifstream& rhs);
11
12 // [ifstream.members], members
13 basic_filebuf<charT, traits>* rdbuf() const;
14
15 bool is_open() const;
16 void open(const char* s, ios_base::openmode mode = ios_base::in);
17 void open(const filesystem::path::value_type* s,
18           ios_base::openmode mode = ios_base::in); // wide systems only; see [fstream.syn]
19 void open(const string& s, ios_base::openmode mode = ios_base::in);
20 void open(const filesystem::path& s, ios_base::openmode mode = ios_base::in);
```

+ `void open(path-view-like s, ios_base::openmode mode = ios_base::in);`

```
1  void close();
2
3  // ...
4  }
```

### In 31.10.4.1 [ifstream.cons] paragraph 2:

```
explicit basic_ifstream(const char* s, ios_base::openmode mode = ios_base::in);
explicit basic_ifstream(const filesystem::path::value_type* s, ios_base::openmode mode = ios_base::in); //wide systems only
+ explicit basic_ifstream(path-view-like s, ios_base::openmode mode = ios_base::in);
```

*Effects:* Initializes the base class with `basic_istream<charT, traits>(addressof(sb))` and `sb` with `basic_filebuf<charT, traits>()`, then calls `rdbuf()->open(s, mode | ios_base::in)`. If that function returns a null pointer, calls `setstate(failbit)`.

### In 31.10.4.3 [ifstream.members] paragraph 4:

```
void open(const char* s, ios_base::openmode mode = ios_base::in);
void open(const filesystem::path::value_type* s, ios_base::openmode mode = ios_base::in); // wide systems only
+ void open(path-view-like s, ios_base::openmode mode = ios_base::in);
```

*Effects:* Calls `rdbuf()->open(s, mode | ios_base::in)`. If that function does not return a null pointer calls `clear()`, otherwise calls `setstate(failbit)` (which may throw `ios_base::failure`)

### In 31.10.5 [ofstream]:

```
1 template<class charT, class traits = char_traits<charT>>
2 class basic_ofstream : public basic_streambuf<charT, traits> {
3     public:
4
5     // ...
6
7     // [ofstream.cons], constructors
8     basic_ofstream();
9     explicit basic_ofstream(const char* s,
10         ios_base::openmode mode = ios_base::in);
11     explicit basic_ofstream(const filesystem::path::value_type* s,
12         ios_base::openmode mode = ios_base::in); // wide systems only; see [fstream.
13         syn]
14     explicit basic_ofstream(const string& s,
15         ios_base::openmode mode = ios_base::in);
```

```
+ explicit basic_ofstream(path-view-like s, ios_base::openmode mode = ios_base::in);
```

```
1     template<class T>
2         explicit basic_ofstream(const T& s, ios_base::openmode mode = ios_base::in);
3     basic_ofstream(const basic_ofstream&) = delete;
4     basic_ofstream(basic_ofstream&& rhs);
5
6     basic_ofstream& operator=(const basic_ofstream&) = delete;
7     basic_ofstream& operator=(basic_ofstream&& rhs);
```



```

8
9 // [ofstream.swap], swap
10 void swap(basic_ofstream& rhs);
11
12 // [ofstream.members], members
13 basic_filebuf<charT, traits>* rdbuf() const;
14
15 bool is_open() const;
16 void open(const char* s, ios_base::openmode mode = ios_base::out);
17 void open(const filesystem::path::value_type* s,
18          ios_base::openmode mode = ios_base::out); // wide systems only; see [fstream.syn]
19 void open(const string& s, ios_base::openmode mode = ios_base::out);
20 void open(const filesystem::path& s, ios_base::openmode mode = ios_base::out);

```

```
+ void open(path-view-like s, ios_base::openmode mode = ios_base::out);
```

```

1 void close();
2
3 // ...
4 }

```

### In 31.10.5.2 [ofstream.cons] paragraph 2:

```

explicit basic_ofstream(const char* s, ios_base::openmode mode = ios_base::out);
explicit basic_ofstream(const filesystem::path::value_type* s, ios_base::openmode mode = ios_base::out); //wide systems only
+ explicit basic_ofstream(path-view-like s, ios_base::openmode mode = ios_base::out);

```

*Effects:* Initializes the base class with `basic_ostream<charT, traits>(addressof(sb))` and `sb` with `basic_filebuf<charT, traits>()`, then calls `rdbuf()->open(s, mode | ios_base::out)`. If that function returns a null pointer, calls `setstate(failbit)`.

### In 31.10.5.4 [ofstream.members] paragraph 3:

```

void open(const char* s, ios_base::openmode mode = ios_base::out);
void open(const filesystem::path::value_type* s, ios_base::openmode mode = ios_base::out); // wide systems only
+ void open(path-view-like s, ios_base::openmode mode = ios_base::out);

```

*Effects:* Calls `rdbuf()->open(s, mode | ios_base::out)`. If that function does not return a null pointer calls `clear()`, otherwise calls `setstate(failbit)` (which may throw `ios_base::failure`).

### In 31.10.6 [fstream]:

```

1  template<class charT, class traits = char_traits<charT>>
2  class basic_fstream : public basic_streambuf<charT, traits> {
3      public:
4
5      // ...
6
7      // [fstream.cons], constructors
8      basic_fstream();
9      explicit basic_fstream(const char* s,
10                          ios_base::openmode mode = ios_base::in);
11     explicit basic_fstream(const filesystem::path::value_type* s,
12                          ios_base::openmode mode = ios_base::in); // wide systems only; see [fstream.
13                                     syn]
14     explicit basic_fstream(const string& s,
15                          ios_base::openmode mode = ios_base::in);

```

+ explicit basic\_fstream(path-view-like s, ios\_base::openmode mode = ios\_base::in);

```

1  template<class T>
2      explicit basic_fstream(const T& s, ios_base::openmode mode = ios_base::in | ios_base::out);
3  basic_fstream(const basic_fstream&) = delete;
4  basic_fstream(basic_fstream&& rhs);
5
6  basic_fstream& operator=(const basic_fstream&) = delete;
7  basic_fstream& operator=(basic_fstream&& rhs);
8
9  // [fstream.swap], swap
10 void swap(basic_fstream& rhs);
11
12 // [fstream.members], members
13 basic_filebuf<charT, traits>* rdbuf() const;
14
15 bool is_open() const;
16 void open(const char* s, ios_base::openmode mode = ios_base::in | ios_base::out);
17 void open(const filesystem::path::value_type* s,
18          ios_base::openmode mode = ios_base::in | ios_base::out); // wide systems only; see [
19                                     fstream.syn]
20 void open(const string& s, ios_base::openmode mode = ios_base::in | ios_base::out);
21 void open(const filesystem::path& s, ios_base::openmode mode = ios_base::in | ios_base::out);

```

+ void open(path-view-like s, ios\_base::openmode mode = ios\_base::in | ios\_base::out);

```

1  void close();
2
3  // ...
4  }

```

### In 31.10.6.2 [fstream.cons] paragraph 2:

```

explicit basic_fstream(const char* s, ios_base::openmode mode = ios_base::in | ios_base::out
);

```

```
explicit basic_fstream(const filesystem::path::value_type* s, ios_base::openmode mode = ios_base::in | ios_base::out); //wide systems only
```

```
+ explicit basic_fstream(path-view-like s, ios_base::openmode mode = ios_base::in | ios_base::out);
```

*Effects:* Initializes the base class with `basic_istream<charT, traits>(addressof(sb))` and `sb` with `basic_filebuf<charT, traits>()`. Then calls `rdbuf()->open(s, mode)`. If that function returns a null pointer, calls `setstate(failbit)`.

### In 31.10.6.3 [fstream.members] paragraph 4:

```
void open(const char* s, ios_base::openmode mode = ios_base::in | ios_base::out);
```

```
void open(const filesystem::path::value_type* s, ios_base::openmode mode = ios_base::in | ios_base::out); //wide systems only
```

```
+ void open(path-view-like s, ios_base::openmode mode = ios_base::in | ios_base::out);
```

*Effects:* Calls `rdbuf()->open(s, mode)`. If that function does not return a null pointer calls `clear()`, otherwise calls `setstate(failbit)` (which may throw `ios_base::failure`).

### 31.12.4 Header `<filesystem>` synopsis [fs.filesystem.syn] 31.12.4.1 Header `<path_view>` synopsis [fs.filesystem.path\_view.syn]

```
namespace std::filesystem {  
  
+ class path_view_component;  
+ class path_view;  
  
+ struct path-view-like { //exposition-only  
+ path_view view;  
+ };  
+ path path-from-binary(span<const byte> data); //exposition-only  
  
+ void swap(path_view_component& lhs, path_view_component& rhs);  
+ size_t hash_value(path_view_component p);  
  
// ...  
  
+ path absolute(path-view-like p);  
+ path absolute(path-view-like p, error_code& ec);  
  
+ path canonical(path-view-like p);  
+ path canonical(path-view-like p, error_code& ec);  
  
+ void copy(path-view-like from, path-view-like to);  
+ void copy(path-view-like from, path-view-like to, error_code& ec)noexcept;
```

```

+ void copy(path-view-like from, path-view-like to, copy_options options);
+ void copy(path-view-like from, path-view-like to, copy_options options, error_code& ec)noexcept
;

+ bool copy_file(path-view-like from, path-view-like to);
+ bool copy_file(path-view-like from, path-view-like to, error_code& ec)noexcept;

+ bool copy_file(path-view-like from, path-view-like to, copy_options options);
+ bool copy_file(path-view-like from, path-view-like to, copy_options options, error_code&
ec)noexcept;

+ void copy_symlink(path-view-like existing_symlink, path-view-like new_symlink);
+ void copy_symlink(path-view-like existing_symlink, path-view-like new_symlink, error_code
& ec)noexcept;

+ bool create_directories(path-view-like p);
+ bool create_directories(path-view-like p, error_code& ec)noexcept;

+ bool create_directory(path-view-like p);
+ bool create_directory(path-view-like p, error_code& ec)noexcept;

+ bool create_directory(path-view-like p, path-view-like attributes);
+ bool create_directory(path-view-like p, path-view-like attributes, error_code& ec)noexcept
;

+ void create_directory_symlink(path-view-like to, path-view-like new_symlink);
+ void create_directory_symlink(path-view-like to, path-view-like new_symlink, error_code&
ec)noexcept;

+ void create_hard_link(path-view-like to, path-view-like new_hard_link);
+ void create_hard_link(path-view-like to, path-view-like new_hard_link, error_code& ec)noexcept
;

+ void create_symlink(path-view-like to, path-view-like new_symlink);
+ void create_symlink(path-view-like to, path-view-like new_symlink, error_code& ec)noexcept
;

+ void current_path(path-view-like p);
+ void current_path(path-view-like p, error_code& ec)noexcept;

+ bool equivalent(path-view-like p1, path-view-like p2);
+ bool equivalent(path-view-like p1, path-view-like p2, error_code& ec)noexcept;

+ bool exists(path-view-like p);
+ bool exists(path-view-like p, error_code& ec)noexcept;

+ uintmax_t file_size(path-view-like p);
+ uintmax_t file_size(path-view-like p, error_code& ec)noexcept;

+ uintmax_t hard_link_count(path-view-like p);
+ uintmax_t hard_link_count(path-view-like p, error_code& ec)noexcept;

```

```

+ bool is_block_file(path-view-like p);
+ bool is_block_file(path-view-like p, error_code& ec)noexcept;

+ bool is_character_file(path-view-like p);
+ bool is_character_file(path-view-like p, error_code& ec)noexcept;

+ bool is_directory(path-view-like p);
+ bool is_directory(path-view-like p, error_code& ec)noexcept;

+ bool is_empty(path-view-like p);
+ bool is_empty(path-view-like p, error_code& ec)noexcept;

+ bool is_fifo(path-view-like p);
+ bool is_fifo(path-view-like p, error_code& ec)noexcept;

+ bool is_other(path-view-like p);
+ bool is_other(path-view-like p, error_code& ec)noexcept;

+ bool is_regular_file(path-view-like p);
+ bool is_regular_file(path-view-like p, error_code& ec)noexcept;

+ bool is_socket(path-view-like p);
+ bool is_socket(path-view-like p, error_code& ec)noexcept;

+ bool is_symlink(path-view-like p);
+ bool is_symlink(path-view-like p, error_code& ec)noexcept;

+ file_time_type last_write_time(path-view-like p);
+ file_time_type last_write_time(path-view-like p, error_code& ec)noexcept;

+ void last_write_time(path-view-like p, file_time_type new_time);
+ void last_write_time(path-view-like p, file_time_type new_time, error_code& ec)noexcept;

+ void permissions(path-view-like p, perms prms, perm_options opts=perm_options::replace);
+ void permissions(path-view-like p, perms prms, error_code& ec)noexcept;
+ void permissions(path-view-like p, perms prms, perm_options opts, error_code& ec);

+ path proximate(path-view-like p, error_code& ec);
+ path proximate(path-view-like p, path-view-like base = current_path());
+ path proximate(path-view-like p, path-view-like base, error_code& ec);

+ path read_symlink(path-view-like p);
+ path read_symlink(path-view-like p, error_code& ec);

+ path relative(path-view-like p, error_code& ec);
+ path relative(path-view-like p, path-view-like base = current_path());
+ path relative(path-view-like p, path-view-like base, error_code& ec);

+ bool remove(path-view-like p);
+ bool remove(path-view-like p, error_code& ec)noexcept;

+ uintmax_t remove_all(path-view-like p);
+ uintmax_t remove_all(path-view-like p, error_code& ec)noexcept;

```

```

+ void rename(path-view-like from, path-view-like to);
+ void rename(path-view-like from, path-view-like to, error_code& ec)noexcept;

+ void resize_file(path-view-like p, uintmax_t size);
+ void resize_file(path-view-like p, uintmax_t size, error_code& ec)noexcept;

+ space_info space(path-view-like p);
+ space_info space(path-view-like p, error_code& ec)noexcept;

+ file_status status(path-view-like p);
+ file_status status(path-view-like p, error_code& ec)noexcept;

bool status_known(file_status s)noexcept; //also defined in <filesystem>

+ file_status symlink_status(path-view-like p);
+ file_status symlink_status(path-view-like p, error_code& ec)noexcept;

+ path weakly_canonical(path-view-like p);
+ path weakly_canonical(path-view-like p, error_code& ec);

```

```

1 }
2
3 namespace std {
4
5 // formatting support

```

```

+ template<class charT> struct formatter<filesystem::path_view_component, charT>;
+ template<class charT> struct formatter<filesystem::path_view, charT>;

```

```

1
2 // hash support

```

```

+ template<> struct hash<filesystem::path_view_component>;
+ template<> struct hash<filesystem::path_view>;

```

```

1 }

```

+ The exposition-only type `path-view-like` is implicitly constructible from any type `T` for which:

- + `std::is_convertible_v<T, path_view>` is true, and
- + `std::is_convertible_v<T, path>` is false

+ and which has a single exposition-only member `view` of type `path_view` which is initialized from the object of type `T`.

+ The exposition-only function `path-from-binary` returns a `path` constructed from a `span<const byte>`.

+ The semantics of this conversion are implementation-defined.

+

[*Note:* The semantics of a binary-to-path conversion are inherently unportable, and may not be possible on all platforms or filesystems. For example, on Windows, if `data.length()` is 16, the returned path could contain the textual representation of a GUID. On POSIX, `data` could be interpreted as an NTBS. – end note]

The free functions taking `path-view-like` and an `error_code` which do not return `path` are `noexcept` as the internal platform-specific allocator in `rendered_path` would return allocation and transcoding failures via `error_code` (usually with `system_category()`) due to the use of a platform-specific function. There is therefore nothing which could throw a C++ exception in this situation.

*Wording note:* The definitions for the function declared in the synopsis above are not provided at this time. All of them delegate to the overload taking a `path`.

### 31.12.4.2 Header `<filesystem>` synopsis [fs.filesystem.syn]

In 31.12.6 [fs.class.path.general] paragraph 6:

```
1 class path {
2 public:
3     using value_type = see below;
4     using string_type = basic_string<value_type>;
5     static constexpr value_type preferred_separator = see below;
6
7     // [fs.enum.path.format], enumeration format
8     enum format;
9
10    // [fs.path.construct], constructors and destructor
11    path() noexcept;
12    path(const path& p);
13    path(path&& p) noexcept;
```

+ explicit path(path-view-like p);

```
1 path(string_type&& source, format fmt = auto_format);
2 template<class Source>
3     path(const Source& source, format fmt = auto_format);
4 template<class InputIterator>
5     path(InputIterator first, InputIterator last, format fmt = auto_format);
6 template<class Source>
7     path(const Source& source, const locale& loc, format fmt = auto_format);
8 template<class InputIterator>
9     path(InputIterator first, InputIterator last, const locale& loc, format fmt = auto_format);
10 ~path();
11
12 // [fs.path.assign], assignments
13 path& operator=(const path& p);
14 path& operator=(path&& p) noexcept;
15 path& operator=(string_type&& source);
16 path& assign(string_type&& source);
17 template<class Source>
18     path& operator=(const Source& source);
19 template<class Source>
```

```

20     path& assign(const Source& source);
21 template<class InputIterator>
22     path& assign(InputIterator first, InputIterator last);
23
24 // [fs.path.append], appends
25 path& operator/=(const path& p);
26 template<class Source>
27     path& operator/=(const Source& source);
28 template<class Source>
29     path& append(const Source& source);
30 template<class InputIterator>
31     path& append(InputIterator first, InputIterator last);
32
33 // [fs.path.concat], concatenation
34 path& operator+=(const path& x);
35 path& operator+=(const string_type& x);
36 path& operator+=(basic_string_view<value_type> x);
37 path& operator+=(const value_type* x);
38 path& operator+=(value_type x);
39 template<class Source>
40     path& operator+=(const Source& x);
41 template<class EcharT>
42     path& operator+=(EcharT x);
43 template<class Source>
44     path& concat(const Source& x);
45 template<class InputIterator>
46     path& concat(InputIterator first, InputIterator last);
47
48 // [fs.path.modifiers], modifiers
49 void clear() noexcept;
50 path& make_preferred();
51 path& remove_filename();
52 path& replace_filename(const path& replacement);

```

```
+ path& replace_filename(path-view-like p);
```

```
1 path& replace_extension(const path& replacement = path());
```

```
+ path& replace_extension(path-view-like p);
```

```

1 void swap(path& rhs) noexcept;
2
3 // [fs.path.nonmember], non-member operators
4 friend bool operator==(const path& lhs, const path& rhs) noexcept;
5 friend strong_ordering operator<=>(const path& lhs, const path& rhs) noexcept;
6
7 friend path operator/(const path& lhs, const path& rhs);

```

```

+ friend path operator/ (path&& lhs, path&& rhs);
+ friend path operator/ (const path& lhs, path-view-like rhs);
+ friend path operator/ (path&& lhs, path-view-like rhs);
+ friend path operator/ (path-view-like lhs, path-view-like rhs);

```

```

1
2 // [fs.path.native.obs], native format observers

```



```
const string_type& native()const+&noexcept;
```

```
+ string_type&& native()&& noexcept;
```

```
1  const value_type* c_str() const noexcept;
2  operator string_type() const;
3
4  template<class EcharT, class traits = char_traits<EcharT>,
5          class Allocator = allocator<EcharT>>
6      basic_string<EcharT, traits, Allocator>
7      string(const Allocator& a = Allocator()) const;
8  std::string  string() const;
9  std::wstring wstring() const;
10 std::u8string u8string() const;
11 std::u16string u16string() const;
12 std::u32string u32string() const;
13
14 // [fs.path.generic.obs], generic format observers
15 template<class EcharT, class traits = char_traits<EcharT>,
16         class Allocator = allocator<EcharT>>
17     basic_string<EcharT, traits, Allocator>
18     generic_string(const Allocator& a = Allocator()) const;
19 std::string  generic_string() const;
20 std::wstring generic_wstring() const;
21 std::u8string generic_u8string() const;
22 std::u16string generic_u16string() const;
23 std::u32string generic_u32string() const;
24
25 // [fs.path.compare], compare
26 int compare(const path& p) const noexcept;
```

```
+ int compare(path-view-like p)const;
```

```
1  int compare(const string_type& s) const;
2  int compare(basic_string_view<value_type> s) const;
3  int compare(const value_type* s) const;
4
5  // [fs.path.decompose], decomposition
6  path root_name() const;
7  path root_directory() const;
8  path root_path() const;
9  path relative_path() const;
10 path parent_path() const;
11 path filename() const;
12 path stem() const;
13 path extension() const;
```

```
+ format formatting()const noexcept;
```

```
1  // [fs.path.query], query
2  bool empty() const noexcept;
3  bool has_root_name() const;
4  bool has_root_directory() const;
5  bool has_root_path() const;
6  bool has_relative_path() const;
7  bool has_parent_path() const;
```

```

8  bool has_filename() const;
9  bool has_stem() const;
10 bool has_extension() const;
11 bool is_absolute() const;
12 bool is_relative() const;
13
14 // [fs.path.gen], generation
15 path lexically_normal() const;
16 path lexically_relative(const path& base) const;

```

```
+ path lexically_relative(path-view-like p) const;
```

```
1 path lexically_proximate(const path& base) const;
```

```
+ path lexically_proximate(path-view-like p) const;
```

```

1
2 // [fs.path.itr], iterators
3 class iterator;
4 using const_iterator = iterator;
5
6 iterator begin() const;
7 iterator end() const;
8
9 // [fs.path.io], path inserter and extractor
10 template<class charT, class traits>
11     friend basic_ostream<charT, traits>&
12     operator<<(basic_ostream<charT, traits>& os, const path& p);
13 template<class charT, class traits>
14     friend basic_istream<charT, traits>&
15     operator>>(basic_istream<charT, traits>& is, path& p);
16 };

```

*Wording note:* The definitions for the member functions declared above are not provided at this time. Their semantics should be relatively obvious, except for `format_formatting()const noexcept`, for which:

+

[Note: If the `path` implementation does not store the formatting with which it was created (all of libstdc++, libc++ and MSVC's implementations currently ignore the parameter entirely), this function ought to return `auto_format`. – end note]

### In 31.12.6.5.1 [fs.path.construct]:

```
+ explicit path(path-view-like p);
```

+ *Effects:* Constructs an object of class `path` by an equivalent call to:

```

1 visit([&p](auto sv) -> path {
2     if constexpr(same_as<remove_cvref_t<decltype(sv)>, span<const byte>>)
3     {
4         return path-from-binary(sv);

```

```

5   }
6   else
7   {
8       return path(sv, p.formatting());
9   }
10 }, p);

```

+ `format formatting()` `const noexcept`;

+ *Returns:* The appropriate path separator format interpretation for the current path's contents.

*Wording note:*

For brevity, I have not described the `path-view-like` added overloads as they are all equivalent to calling the `path` overload with a path constructed from the path view. Obviously implementations can be more efficient here by avoiding a dynamic memory allocation in a temporarily constructed path.

## Class `path_view_component` [`fs.path_view_component`]

An object of class `path_view_component` refers to a source of data from which a filesystem path can be derived. To avoid confusion, in the remainder of this section this source of data shall be called *the backing data*.

Any operation that invalidates a pointer within the range of that backing data invalidates pointers, iterators and references returned by `path_view_component`.

`path_view_component` is trivially copyable.

The complexity of `path_view_component` member functions is  $O(1)$  unless otherwise specified.

```

1 namespace std::filesystem {
2     class path_view_component {
3     public:
4         using size_type = /* implementation defined */;
5         static constexpr path::value_type preferred_separator = path::preferred_separator;
6         static constexpr size_t default_internal_buffer_size = /* implementation defined */;
7
8         using format = path::format;
9
10        enum termination {
11            null_terminated,
12            unterminated
13        };
14
15        template<class T>
16        /* implementation defined */ default_rendered_path_allocator = /* implementation defined */;
17
18        // Constructors and destructor
19        constexpr path_view_component() noexcept;
20
21        path_view_component(path_view_component, format fmt) noexcept;
22

```

```

23 path_view_component(const path &p) noexcept;
24 template<class CharT>
25 constexpr path_view_component(const basic_string<CharT>& s,
26                               format fmt = path::binary_format) noexcept;
27
28 template<class CharT>
29 constexpr path_view_component(const CharT* b, size_type l, enum termination zt,
30                               format fmt = path::binary_format) noexcept;
31 constexpr path_view_component(const byte* b, size_type l, enum termination zt) noexcept;
32
33 template<class CharT>
34 constexpr path_view_component(const CharT* b, format fmt = path::binary_format) noexcept;
35 constexpr path_view_component(const byte* b) noexcept;
36
37 template<class CharT>
38 constexpr path_view_component(basic_string_view<CharT> b, enum termination zt,
39                               format fmt = path::binary_format) noexcept;
40 constexpr path_view_component(span<const byte> b, enum termination zt) noexcept;
41
42 template<class It, class End>
43 constexpr path_view_component(It b, End e, enum termination zt,
44                               format fmt = path::binary_format) noexcept;
45 template<class It, class End>
46 constexpr path_view_component(It b, End e, enum termination zt) noexcept;
47
48 constexpr path_view_component(const path_view_component&) = default;
49 constexpr path_view_component(path_view_component&&) = default;
50 constexpr ~path_view_component() = default;
51
52 // Assignments
53 constexpr path_view_component &operator=(const path_view_component&) = default;
54 constexpr path_view_component &operator=(path_view_component&&) = default;
55
56 // Modifiers
57 constexpr void swap(path_view_component& o) noexcept;
58
59 // Query
60 constexpr bool empty() const noexcept;
61 constexpr size_type native_size() const noexcept;
62 constexpr format formatting() const noexcept;
63 constexpr bool has_null_termination() const noexcept;
64 constexpr enum termination termination() const noexcept;
65 constexpr bool has_stem() const noexcept;
66 constexpr bool has_extension() const noexcept;
67
68 constexpr path_view_component stem() const noexcept;
69 constexpr path_view_component extension() const noexcept;
70
71 // Comparison
72 template<class T = typename path::value_type,
73         class Allocator = default_rendered_path_allocator<T>,
74         size_type InternalBufferSize = default_internal_buffer_size>
75 constexpr int compare(path_view_component p) const;
76
77 // Conversion
78 template<enum path_view_component::termination Termination,

```

```

79         class T = typename path::value_type,
80         class Allocator = default_rendered_path_allocator<T>,
81         size_type InternalBufferSize = default_internal_buffer_size>
82     class rendered_path;
83
84     // Conversion convenience
85     template <class T = typename path::value_type,
86             class Allocator = default_rendered_path_allocator<T>,
87             size_type InternalBufferSize = default_internal_buffer_size>
88     constexpr rendered_path<termination::null_terminated,
89             T, Allocator, _internal_buffer_size>
90     render_null_terminated(Allocator allocate = Allocator()) const;
91
92     template <class T = typename path::value_type,
93             class Allocator = default_rendered_path_allocator<T>,
94             size_type InternalBufferSize = default_internal_buffer_size>
95     constexpr rendered_path<termination::unterminated,
96             T, Allocator, _internal_buffer_size>
97     render_unterminated(Allocator allocate = Allocator()) const;
98
99
100 private:
101     union
102     {
103         const byte* bytestr_{nullptr}; // exposition only
104         const char* charstr_; // exposition only
105         const wchar_t* wcharstr_; // exposition only
106         const char8_t* char8str_; // exposition only
107         const char16_t* char16str_; // exposition only
108     };
109     size_type length_{0}; // exposition only
110     uint16_t null_terminated_ : 1; // exposition only
111     uint16_t is_bytestr_ : 1; // exposition only
112     uint16_t is_charstr_ : 1; // exposition only
113     uint16_t is_wcharstr_ : 1; // exposition only
114     uint16_t is_char8str_ : 1; // exposition only
115     uint16_t is_char16str_ : 1; // exposition only
116     format format_{format::unknown}; // exposition only
117 };
118 /* Note to be removed before LWG: if your platform has a maximum path size
119 which fits inside a uint32_t, it is possible to pack path views
120 into 2 * sizeof(void*), which can be returned in CPU registers on
121 x64 Itanium ABI.
122 */
123 static_assert(std::is_trivially_copyable_v<path_view_component>); // to be removed before LWG
124 static_assert(sizeof(path_view_component) == 2 * sizeof(void*)); // to be removed before LWG
125
126 // Comparison
127 inline constexpr bool operator==(path_view_component a, path_view_component b) noexcept;
128 inline constexpr strong_ordering operator<=>(path_view_component a, path_view_component b) noexcept;
129
130 // Disabled comparisons
131 template<class CharT>
132 inline constexpr bool operator==(path_view_component, const CharT*) = delete(/* implementation
133 defined message */);
134 template<class CharT>

```

```

134 inline constexpr bool operator==(path_view_component, basic_string_view<CharT>) = delete(/*
      implementation defined message */);
135 inline constexpr bool operator==(path_view_component, const byte*) = delete(/* implementation
      defined message */);
136 inline constexpr bool operator==(path_view_component, span<const byte>) = delete(/* implementation
      defined message */);
137
138 template<class CharT>
139 inline constexpr auto operator<=>(path_view_component, const CharT*) = delete(/* implementation
      defined message */);
140 template<class CharT>
141 inline constexpr auto operator<=>(path_view_component, basic_string_view<CharT>) = delete(/*
      implementation defined message */);
142 inline constexpr auto operator<=>(path_view_component, const byte*) = delete(/* implementation
      defined message */);
143 inline constexpr auto operator<=>(path_view_component, span<const byte>) = delete(/* implementation
      defined message */);
144
145 template<class CharT>
146 inline constexpr bool operator==(const CharT*, path_view_component) = delete(/* implementation
      defined message */);
147 template<class CharT>
148 inline constexpr bool operator==(basic_string_view<CharT>, path_view_component) = delete(/*
      implementation defined message */);
149 inline constexpr bool operator==(const byte*, path_view_component) = delete(/* implementation
      defined message */);
150 inline constexpr bool operator==(span<const byte>, path_view_component) = delete(/* implementation
      defined message */);
151
152 template<class CharT>
153 inline constexpr auto operator<=>(const CharT*, path_view_component) = delete(/* implementation
      defined message */);
154 template<class CharT>
155 inline constexpr auto operator<=>(basic_string_view<CharT>, path_view_component) = delete(/*
      implementation defined message */);
156 inline constexpr auto operator<=>(const byte*, path_view_component) = delete(/* implementation
      defined message */);
157 inline constexpr auto operator<=>(span<const byte>, path_view_component) = delete(/* implementation
      defined message */);
158
159 // Hash value
160 size_t hash_value(path_view_component v) noexcept;
161
162 // Visitation
163 template<class F>
164 inline constexpr auto visit(F &&f, path_view_component v);
165
166 // Output
167 template<class charT, class traits>
168 basic_ostream<charT, traits>& operator<<(basic_ostream<charT, traits>& os, path_view_component v);
169 }

```

The value of the `default_internal_buffer_size` member is an implementation chosen value for the default internal character buffer held within a `path_view_component::rendered_path` instance, which is usually instantiated onto the stack. It ought to be defined to a little more than the typical length

of filesystem path on that platform<sup>3</sup>.

Enumeration `format` determines how, and whether, to interpret path separator characters within path views' backing data:

- `unknown` may cause a run time diagnostic if path components need to be delineated. Depends on operation.
- `native_format` causes only the native path separator character to delineate path components.
- `generic_format` causes only the generic path separator character ('/') to delineate path components.
- `binary_format` causes no delineation of path components at all in the backing data.
- `auto_format` causes *both* the native and generic path separators to delineate path components (and backing data may contain a mix of both).

Enumeration `termination` allows users to specify whether the backing data has a zeroed value after the end of the supplied input.

`default_rendered_path_allocator<T>` is a type possibly tagging the internal selection of an implementation defined allocator.

### Construction and assignment [`fs.path_view_component.cons`]

```
1 constexpr path_view_component() noexcept;
```

*Effects:* Constructs an object of class `path_view_component` which is empty.

*Ensures:* `empty() == true` and `formatting() == format::unknown`.

```
1 path_view_component(path_view_component, format fmt) noexcept;
```

*Effects:* Constructs an object of class `path_view_component` which refers to the same backing data as the input path view component, but with different interpretation of path separators.

*Ensures:* `formatting() == fmt`.

```
1 path_view_component(const path &p) noexcept;
```

*Effects:* Constructs an object of class `path_view_component` which refers to a zero terminated contiguous sequence of `path::value_type` which begins at `p.c_str()` and continues for `p.native().size()` items.

*Ensures:* `formatting() == p.formatting()` and `termination() == null_terminated`.

---

<sup>3</sup>After much deliberation, LEWG chose 1,024 codepoints as a reasonable suggested default for most platforms.

```

1  template<class CharT>
2  constexpr path_view_component(const basic_string<CharT>& s,
3                               format fmt = path::binary_format) noexcept;

```

*Constraints:* CharT is any one of: char, wchar\_t, char8\_t, char16\_t.

*Effects:* Constructs an object of class path\_view\_component which refers to [ s.data(), s.data()+s.size()).

*Ensures:* formatting()== fmt and termination()== null\_terminated.

```

1  template<class CharT>
2  constexpr path_view_component(const CharT* b, size_type l, enum termination zt,
3                               format fmt = path::binary_format) noexcept;

```

*Constraints:* CharT is any one of: char, wchar\_t, char8\_t, char16\_t.

*Expects:* If zt is null\_terminated, then [b, b + l] is a valid range and b[l] == CharT(0); otherwise [b, b + l) is a valid range.

*Effects:* Constructs an object of class path\_view\_component which refers to a contiguous sequence of one of char, wchar\_t, char8\_t or char16\_t which begins at b and continues for l items.

*Ensures:* formatting()== fmt and termination()== zt.

```

1  constexpr path_view_component(const byte* b, size_type l, enum termination zt) noexcept;

```

*Expects:* If zt is null\_terminated, then [b, b + l] is a valid range and b[l] == CharT(0); otherwise [b, b + l) is a valid range.

*Effects:* Constructs an object of class path\_view\_component which refers to a contiguous sequence of byte which begins at b and continues for l items.

*Ensures:* formatting()== format::binary\_format and termination()== zt.

```

1  template<class CharT>
2  constexpr path_view_component(const CharT* b, format fmt = path::binary_format) noexcept;

```

*Constraints:* CharT is any one of: char, wchar\_t, char8\_t, char16\_t.

*Expects:* [b, b + char\_traits<CharT>::length(b)] is a valid range.

*Effects:* Equivalent to path\_view\_component(b, char\_traits<CharT>::length(b), fmt).

*Ensures:* formatting()== fmt and termination()== null\_terminated.

*Complexity:* O(char\_traits<CharT>::length(b)).

```

1  constexpr path_view_component(const byte* b) noexcept;

```



*Expects:* Let as if `e = static_cast<const byte *>(memchr(b, 0))`, then `[b, e]` is a valid range.

*Effects:* Equivalent to `path_view_component(b, (size_type)(e - b))`, if `memchr` were a `constexpr` available function.

*Ensures:* `formatting() == format::binary_format` and `termination() == null_terminated`.

*Complexity:* `O(e - b)`.

*Remarks:* If the consumer of path view components interprets byte input as a fixed length binary key, then it will pass the byte pointer as-is to the relevant system call. If the byte range has an incorrect length for the destination, the behaviour is unspecified.

```
1     template<class CharT>
2     constexpr path_view_component(basic_string_view<CharT> b, enum termination zt,
3                               format fmt = path::binary_format) noexcept;
```

*Constraints:* `CharT` is any one of: `char`, `wchar_t`, `char8_t`, `char16_t`; if `zt` is `null_terminated`, then `b.data()[b.size()] == CharT(0)`.

*Effects:* Equivalent to `path_view_component(b.data(), b.size(), zt, fmt)`.

*Ensures:* `formatting() == fmt` and `termination() == zt`.

```
1     constexpr path_view_component(span<const byte> b, enum termination zt) noexcept;
```

*Constraints:* If `zt` is `null_terminated`, then `b.data()[b.size()] == byte(0)`.

*Effects:* Equivalent to `path_view_component(b.data(), b.size(), zt)`.

*Ensures:* `formatting() == format::binary_format` and `termination() == zt`.

```
1     template<class It, class End>
2     constexpr path_view_component(It b, End e, enum termination zt,
3                               format fmt = path::binary_format) noexcept;
```

*Constraints:*

1. `It` satisfies `contiguous_iterator`.
2. `End` satisfies `sized_sentinel_for<It>`.
3. `iter_value_t<It>` is any one of: `char`, `wchar_t`, `char8_t`, `char16_t`.
4. `is_convertible_v<End, size_type>` is false.
5. If `zt` is `null_terminated`, then `*e == X(0)`.

*Expects:*

1. If `zt` is `null_terminated`, then `[b, e]` is a valid range, otherwise `[b, e)` is a valid range.
2. `It` models `contiguous_iterator`.

3. End models `sized_sentinel_for<It>`.

*Effects:* Equivalent to `path_view_component(to_address(begin), end - begin, zt, fmt)`.

*Ensures:* `formatting()== fmt` and `termination()== zt`.

```
1  template<class It, class End>
2  constexpr path_view_component(It b, End e, enum termination zt) noexcept;
```

*Constraints:*

1. It satisfies `contiguous_iterator`.
2. End satisfies `sized_sentinel_for<It>`.
3. `iter_value_t<It>` is `byte`.
4. `is_convertible_v<End, size_type>` is false.
5. If `zt` is `null_terminated`, then `*e == byte(0)`.

*Expects:*

1. If `zt` is `null_terminated`, then `[b, e]` is a valid range, otherwise `[b, e)` is a valid range.
2. It models `contiguous_iterator`.
3. End models `sized_sentinel_for<It>`.

*Effects:* Equivalent to `path_view_component(to_address(begin), end - begin, zt)`.

*Ensures:* `formatting()== format::binary_format` and `termination()== zt`.

## Modifiers [`fs.path_view_component.modifiers`]

```
1  constexpr void swap(path_view_component& o) noexcept;
```

*Effects:* Exchanges the values of `*this` and `o`.

## Observers [`fs.path_view_component.observers`]

```
1  constexpr bool empty() const noexcept;
```

*Returns:* True if `native_size()== 0`.

```
1  constexpr size_type native_size() const noexcept;
```

*Returns:* The number of codepoints, or bytes, with which the path view component was constructed.

```
1 constexpr format formatting() const noexcept;
```

*Returns:* The formatting with which the path view component was constructed.

```
1 constexpr bool has_null_termination() const noexcept;
```

*Returns:* True if the path view component was constructed with zero termination.

```
1 constexpr enum termination termination() const noexcept;
```

*Returns:* The zero termination with which the path view component was constructed.

```
1 constexpr bool has_stem() const noexcept;
```

*Returns:* True if `stem()` return a non-empty path view component.

*Complexity:* `O(native_size())`.

```
1 constexpr bool has_extension() const noexcept;
```

*Returns:* True if `extension()` return a non-empty path view component.

*Complexity:* `O(native_size())`.

```
1 constexpr path_view_component stem() const noexcept;
```

*Returns:* Let `s` refer to one element of backing data after the last separator element `sep` as interpreted by `formatting()` in the path view component, otherwise then to the first element in the path view component; let `e` refer to the last period within `[s + 1, native_size())` unless `[s, native_size())` is `'..'`, otherwise then to one past the last element in the path view component; returns the portion of the path view component matching `[s, e)`.

*Complexity:* `O(native_size())`.

*Remarks:* The current normative wording for `path::stem()` is unclear how to handle `"/foo/bar/.."`, so here `stem()` returns `'..'` and `extension()` returns `''` in this circumstance.

```
1 constexpr path_view_component extension() const noexcept;
```

*Returns:* Let `s` refer to one element of backing data after the last separator element `sep` as interpreted by `formatting()` in the path view component, otherwise then to the first element in the path view component; let `e` refer to the last period within `[s + 1, native_size())` unless `[s, native_size())`

is '.', otherwise then to one past the last element in the path view component; returns the portion of the path view component matching [e, native\_size()).

*Complexity:* O(native\_size()).

```
1     template<class T = typename path::value_type,
2             class Allocator = default_rendered_path_allocator<T>,
3             size_type InternalBufferSize = default_internal_buffer_size>
4     constexpr int compare(path_view_component p) const;
```

*Constraints:* T is any one of: char, wchar\_t, char8\_t, char16\_t, byte; Allocator is either its defaulted internal tag type, or meets Cpp17Allocator requirements.

*Effects:*

- If T is byte, the comparison of the two backing data ranges is implemented as a byte comparison equivalent to memcmp.
- Otherwise the comparison is equivalent to:

```
1     path_view_component::rendered_path<T, Allocator, InternalBufferSize> zpath1(*this), zpath2(p
2     );
3     path path1(zpath1.buffer, zpath1.length, this->formatting()), path2(zpath2.buffer, zpath2.
4     length, p.formatting());
5     path1.compare(path2);
```

*Complexity:* O(native\_size()).

*Remarks:* The above wording is intended to retain an important source of optimisation whereby implementations do not actually have to construct a path\_view\_component::rendered\_path nor a path from those buffers e.g. if the backing data for both \*this and p are of the same encoding, the two backing data ranges can be compared directly (ignoring multiple path separators etc), if and only if the same comparison result would occur if both buffers were converted to path and those paths compared.

```
1     template <class T = typename path::value_type,
2             class Allocator = default_rendered_path_allocator<T>,
3             size_type InternalBufferSize = default_internal_buffer_size>
4     constexpr rendered_path<termination::null_terminated,
5             T, Allocator, _internal_buffer_size>
6     render_null_terminated(Allocator allocate = Allocator()) const;
```

*Returns:* rendered\_path<termination::null\_terminated, T, Allocator, \_internal\_buffer\_size>(\*this, allocate);

```
1     template <class T = typename path::value_type,
2             class Allocator = default_rendered_path_allocator<T>,
3             size_type InternalBufferSize = default_internal_buffer_size>
4     constexpr rendered_path<termination::unterminated,
5             T, Allocator, _internal_buffer_size>
6     render_unterminated(Allocator allocate = Allocator()) const;
```

*Returns:* `rendered_path<termination::unterminated, T, Allocator, _internal_buffer_size>(*this, allocate);`

## Class `path_view_component::rendered_path` [`fs.path_view_component.rendered_path`]

```
1 namespace std::filesystem {
2     template<enum path_view_component::termination Termination,
3             class T = typename path::value_type,
4             class Allocator = path_view_component::default_rendered_path_allocator<T>,
5             size_type InternalBufferSize = path_view_component::default_internal_buffer_size>
6     class path_view_component::rendered_path {
7     public:
8         using value_type = const T;
9         using pointer = const T*;
10        using const_pointer = const T*;
11        using reference = const T&;
12        using const_reference = const T&;
13        using iterator = span<value_type>::iterator;
14        using const_iterator = span<value_type>::const_iterator;
15        using reverse_iterator = span<value_type>::reverse_iterator;
16        using const_reverse_iterator = span<value_type>::const_reverse_iterator;
17        using size_type = span<value_type>::size_type;
18        using difference_type = span<value_type>::difference_type;
19        using allocator_type = Allocator; /* not present if default_rendered_path_allocator tag type was
20        used */
21
22    public:
23        // constructors and destructor
24        rendered_path() noexcept;
25        ~rendered_path();
26
27        constexpr rendered_path(path_view_component v, Allocator allocate = Allocator());
28
29        rendered_path(const rendered_path&) = delete;
30        rendered_path(rendered_path&& o) noexcept;
31
32        // assignment
33        rendered_path &operator=(const rendered_path&) = delete;
34        rendered_path &operator=(rendered_path&&) noexcept;
35
36        // iteration
37        constexpr iterator begin() noexcept;
38        constexpr const_iterator begin() const noexcept;
39        constexpr const_iterator cbegin() const noexcept;
40        constexpr iterator end() noexcept;
41        constexpr const_iterator end() const noexcept;
42        constexpr const_iterator cend() const noexcept;
43        constexpr reverse_iterator rbegin() noexcept;
44        constexpr const_reverse_iterator rbegin() const noexcept;
45        constexpr const_reverse_iterator crbegin() const noexcept;
46        constexpr reverse_iterator rend() noexcept;
47        constexpr const_reverse_iterator rend() const noexcept;
48        constexpr const_reverse_iterator crend() const noexcept;
```

```

49 // access
50 constexpr reference operator[](size_type idx) noexcept;
51 constexpr const_reference operator[](size_type idx) const noexcept;
52 constexpr reference at(size_type idx);
53 constexpr const_reference at(size_type idx) const;
54 constexpr reference front() noexcept;
55 constexpr const_reference front() const noexcept;
56 constexpr reference back() noexcept;
57 constexpr const_reference back() const noexcept;
58 constexpr pointer data() noexcept;
59 constexpr const_pointer data() noexcept;
60 constexpr size_type size() const noexcept;
61 constexpr size_type length() const noexcept;
62 constexpr size_type max_size() const noexcept;
63 constexpr bool empty() noexcept;
64
65 constexpr allocator_type get_allocator() const noexcept; /* not present if
    default_rendered_path_allocator tag type was used */
66
67 constexpr size_t capacity() const noexcept;
68 constexpr bool references_source() const noexcept;
69
70 constexpr span<const value_type> as_span() const noexcept;
71
72 constexpr const_pointer c_str() const noexcept; // available only if null_terminated
73
74 private:
75     span<const value_type> _ref; // exposition only
76     size_t bytes_to_delete_{0}; // exposition only
77     Allocator allocator_; // exposition only
78     value_type buffer_[internal_buffer_size]{}; // exposition only
79
80     /* To be removed before LWG:
81
82     Note that if the internal buffer is the final item in the structure,
83     the major C++ compilers shall, if they can statically prove that
84     the buffer will never be used, entirely eliminate it from runtime
85     codegen. This can happen quite frequently during aggressive
86     inlining if the backing data is a string literal.
87     */
88     };
89 }

```

*Constraints:* `T` is any one of: `char`, `wchar_t`, `char8_t`, `char16_t`, `byte`; `Allocator` is either its defaulted internal tag type, or meets *Cpp17Allocator* requirements.

Class `path_view_component::rendered_path` is a mechanism for rendering a path view component's backing data into a buffer, optionally reencoded, optionally zero terminated. It is expected to be, in most cases, much more efficient than constructing a `path` from visiting the backing data, however unlike `path` it can also target non-`path::value_type` consumers of filesystem paths e.g. other programming languages or archiving libraries.

The lifetime of the contained data in a `path_view_component::rendered_path` is tied to the backing data of the `path_view_component` used to construct it, and not to the lifetime of the `path_view_component`

itself.

It is important to note that the consumer of path view components determines the interpretation of path view components, not class `path_view_component::rendered_path` nor `path`. For example, if the backing data is unencoded bytes, a consuming implementation might choose to use a binary key API to open filesystem content instead of a path based API whose input comes from `path_view_component::rendered_path` or `path` i.e. APIs consuming path view components may behave differently if the backing data is in one format, or another.

[*Note:* For example, Microsoft Windows has system APIs which can open a file by binary key specified in the `FILE_ID_DESCRIPTOR` structure. Some POSIX implementations support the standard SNIA NVMe key-value API for storage devices. **IMPORTANT:** If a consuming implementation expects to, in the future, interpret byte backing data differently e.g. it does not support binary key lookup on a filesystem now, but may do so in the future, **it ought to reject byte backed path view components now** with an appropriate error instead of utilising the `rendered_path` byte passthrough described below. – end note]

After construction, an object of class `path_view_component::rendered_path` will have members `data()` and `size()` set as follows: `data()` will point at an optionally zero terminated array of `value_type` of length `size()`, the count of which excludes any zero termination. Furthermore, if `Termination` is `null_terminated`, `c_str()` additionally becomes available.

### Construction [fs.path\_view\_component.rendered\_path.cons]

```
1 ~rendered_path();
```

*Effects:* If during construction a dynamic memory allocation was required, that is released using the `Allocator` instance which was supplied during construction, or the internal platform-specific allocator if appropriate.

```
1 constexpr rendered_path(path_view_component v, Allocator allocate = Allocator());
```

*Effects:*

- If `value_type` is `byte`, `size()` will return `v.native_size()`. If `termination` is `null_terminated` and `v.termination()` is `unterminated`:
  - If `size() < internal_buffer_size - 1`:
    - \* `data()` returns `buffer_`, the bytes of the backing data are copied into `buffer_`, and a zero valued byte is appended.
  - else:

- \* `allocate.allocate(length + 1)` is performed to yield the value returned by `data()`, the bytes of the backing data are copied into `data()`, and a zero valued byte is appended.

else:

- `data()` returns the backing data.

- If the backing data is `byte` and `value_type` is not `byte`, `size()` will return `v.native_size()/sizeof(value_type)`. If `termination` is `null_terminated`, and either `(v.native_size()+v.has_null_termination())!= (size()+ 1)* sizeof(value_type)` is true or `v.termination()` is `unterminated`:

- If `size()< internal_buffer_size - 1`:

- \* `data()` returns `buffer_`, the bytes of the backing data are copied into `buffer_`, and a zero valued `value_type` is appended.

else:

- \* `allocate.allocate(length + 1)` is performed to yield the value returned by `data()`, the bytes of the backing data are copied into `data()`, and a zero valued `value_type` is appended.

else:

- `data()` returns the backing data.

*Remarks:* The `(v.native_size()+ v.has_null_termination())!= (size()+ 1)* sizeof(value_type)` is to enable passthrough of byte input to `wchar_t` output by passing in an uneven sized byte input marked as zero terminated, whereby if the zero terminated byte is added into the input, the total sum of bytes equals exactly the number of bytes which the zero terminated output buffer would occupy. The inferred promise here is that the code which constructed the path view with raw bytes and zero termination has appropriately padded the end of the buffer with the right number of zero bytes to make up a null terminated `wchar_t`.

- If the backing data and `value_type` have the same bit-for-bit encoding in the wide sense (e.g. if the narrow system encoding `char` is considered to be UTF-8, it is considered the same encoding as `char8_t`; similarly if the wide system encoding `wchar_t` is considered to be UTF-16, it is considered the same encoding as `char16_t`, and so on), `size()` will return `v.native_size()`. If `termination` is `null_terminated` and `v.termination()` is `unterminated`, or depending on the value of `v.formatting()` the backing data contains any generic path separators and the generic path separator is not the native path separator:

- If `size()< internal_buffer_size - 1`:

- \* `data()` returns `buffer_`, the code points of the backing data are copied into `buffer_`, replacing any generic path separators with native path separators if `v.formatting()` allows that, and a zero valued `value_type` is appended.

else:



\* `allocate.allocate(length + 1)` is performed to yield the value returned by `data()`, the code points of the backing data are copied into `data()`, replacing any generic path separators with native path separators if `v.formatting()` allows that, and a zero valued `value_type` is appended.

else:

– `data()` returns the backing data.

- Otherwise, a reencoding of the backing data into `value_type` shall be performed, replacing any generic path separators with native path separators if `v.formatting()` allows that, zero `value_type` terminating the reencoded buffer if `termination` is `null_terminated`. `data()` shall return that reencoded path, and `size()` shall be the number of elements output, excluding any zero termination appended.

## Observers [fs.rendered\_path.obs]

[*Note:* The vast majority of the observers replicate those of `span` and so are not described further here. The reason `span` was chosen over `basic_string_view` is because the rendered path could be binary. – end note]

```
1 constexpr allocator_type get_allocator() const noexcept; /* not present if
   default_rendered_path_allocator tag type was used */
```

*Constraints:* `Allocator` meets *Cpp17Allocator* requirements.

*Returns:* The allocator associated with the rendered path.

```
1 constexpr size_t capacity() const noexcept;
```

*Returns:* The maximum number of rendered items which could be stored in this rendered path instance without causing a new dynamic memory allocation.

```
1 constexpr bool references_source() const noexcept;
```

*Returns:* True if this rendered path references backing data elsewhere.

```
1 constexpr span<const value_type> as_span() const noexcept;
```

*Effects:* Returns a span representing the rendered path.

```
1 constexpr const_pointer c_str() const noexcept; // available only if null_terminated
```

*Constraints:* `Termination` is `termination::null_terminated`.

*Returns:* The same value as `data()`.

## Non-member comparison functions [fs.path\_view\_component.comparison]

```
1 inline constexpr bool operator==(path_view_component a, path_view_component b) noexcept;
```

*Returns:* True if: (i) the backing encoding is equal (ii) the size of backing representation is equal (iii) a comparison equivalent to `memcmp` finds equal bit representations of the backing bytes.

[*Note:* This is intentionally a ‘shallow’ equality comparison intended for use in maps etc, it doesn’t do expensive `compare()`. – end note]

```
1 inline constexpr strong_ordering operator<=>(path_view_component a, path_view_component b) noexcept;
```

*Effects:* Ordering is considered in this order: (i) `byte` encoding (ii) `char` encoding (iii) `wchar_t` encoding (iv) `char8_t` encoding (v) `char16_t` encoding (vi) size of backing representation (vii) a bit comparison ordering equivalent to `memcmp`.

[*Note:* This is intentionally a ‘shallow’ comparison intended for use in maps etc, it doesn’t do expensive `compare()`. – end note]

```
1 template<class CharT>
2 inline constexpr bool operator==(path_view_component, const CharT*) = delete(/* implementation
   defined message */);
3 template<class CharT>
4 inline constexpr bool operator==(path_view_component, basic_string_view<CharT>) = delete(/*
   implementation defined message */);
5 inline constexpr bool operator==(path_view_component, const byte*) = delete(/* implementation
   defined message */);
6 inline constexpr bool operator==(path_view_component, span<const byte>) = delete(/* implementation
   defined message */);
7
8 template<class CharT>
9 inline constexpr auto operator<=>(path_view_component, const CharT*) = delete(/* implementation
   defined message */);
10 template<class CharT>
11 inline constexpr auto operator<=>(path_view_component, basic_string_view<CharT>) = delete(/*
   implementation defined message */);
12 inline constexpr auto operator<=>(path_view_component, const byte*) = delete(/* implementation
   defined message */);
13 inline constexpr auto operator<=>(path_view_component, span<const byte>) = delete(/* implementation
   defined message */);
14
15 template<class CharT>
16 inline constexpr bool operator==(const CharT*, path_view_component) = delete(/* implementation
   defined message */);
17 template<class CharT>
18 inline constexpr bool operator==(basic_string_view<CharT>, path_view_component) = delete(/*
   implementation defined message */);
19 inline constexpr bool operator==(const byte*, path_view_component) = delete(/* implementation
   defined message */);
20 inline constexpr bool operator==(span<const byte>, path_view_component) = delete(/* implementation
   defined message */);
21
22 template<class CharT>
23 inline constexpr auto operator<=>(const CharT*, path_view_component) = delete(/* implementation
   defined message */);
```

```

24  template<class CharT>
25  inline constexpr auto operator<=>(basic_string_view<CharT>, path_view_component) = delete(/*
    implementation defined message */);
26  inline constexpr auto operator<=>(const byte*, path_view_component) = delete(/* implementation
    defined message */);
27  inline constexpr auto operator<=>(span<const byte>, path_view_component) = delete(/* implementation
    defined message */);

```

*Effects:* Comparing string literals, string views, byte literals or byte views, against a path view component is deleted. A diagnostic explaining that `.compare()` or `visit()` ought to be used instead is suggested.

## Non-member functions [fs.path\_view\_component.hash]

```

1  size_t hash_value(path_view_component v) noexcept;

```

*Returns:* A hash value for the path `v`. If for two path view components, `p1 == p2` then `hash_value(p1) == hash_value(p2)`.

```

1  template<class F>
2  inline constexpr auto visit(F &&f, path_view_component v);

```

*Constraints:* All of these are true:

- invocable<F, basic\_string\_view<char>>.
- invocable<F, basic\_string\_view<wchar\_t>>.
- invocable<F, basic\_string\_view<char8\_t>>.
- invocable<F, basic\_string\_view<char16\_t>>.
- invocable<F, span<const byte>>.

*Effects:* The callable `f` is invoked with a `basic_string_view<CharT>` if the backing data has a character encoding, otherwise it is invoked with a `span<const byte>` with the backing bytes.

*Returns:* Whatever `F` returns.

```

1  template<class charT, class traits>
2  basic_ostream<charT, traits>& operator<<(basic_ostream<charT, traits>& os, path_view_component v);

```

*Effects:* Equivalent to:

```

1  template<class charT, class traits>
2  basic_ostream<charT, traits>& operator<<(basic_ostream<charT, traits>& os, path_view_component v)
3  {
4  return visit([&os, &v](auto sv) -> basic_ostream<charT, traits>& {
5      using input_type = remove_cvref_t<decltype(sv)>;
6      using output_type = basic_ostream<charT, traits>;

```

```

7
8   if constexpr(same_as<input_type, span<const byte>>)
9   {
10      /* Handle byte encoded filesystem paths however the
11         implementation handles them. For example, Microsoft Windows
12         requires the following textualisation for
13         FILE_ID_DESCRIPTOR.ObjectId keys which are guids:
14
15         "{7ecf65a0-4b78-5f9b-e77c-8770091c0100}"
16
17         This is a valid filename in NTFS with special semantics:
18         OpenFileById() is used instead if you pass it into
19         CreateFile().
20
21         Otherwise some textual representation which is not
22         a possible valid textual path is suggested.
23         */
24      return os << quoted(\emph{path-from-binary}(sv).string<typename output_type::char_type>());
25   }
26   else
27   {
28      // Possibly reencode to ostream's character type
29      path_view_component::rendered_path<typename output_type::char_type> zbuff(v, path_view_component
30      ::unterminated);
31      return os << quoted(basic_string_view<typename output_type::char_type>(zbuff.buffer, zbuff.
32      length));
33   }
34 }, v);
35 }

```

*Returns:* `os`.

## Class `path_view` [`fs.path_view`]

An object of class `path_view` is a `path_view_component` which has additional functionality:

- It is an iterable sequence of `path_view_component` returning subsets of the path view.
- It has additional member functions implementing corresponding functionality from `path`.
- Constructing a `path_view_component` for a piece of backing data defaults to `binary_format` interpretation of path separators, whereas constructing a `path_view` for a piece of backing data defaults to `auto_format` interpretation of path separators. `path_view_component`'s yielded from iteration of `path_view` have `binary_format` interpretation of path separators.

`path_view` is trivially copyable.

The complexity of `path_view` member functions is  $O(1)$  unless otherwise specified.

```

1 namespace std::filesystem {
2   class path_view : public path_view_component {
3   public:
4     using const_iterator = /* implementation defined */;

```

```

5     using iterator = /* implementation defined */;
6     using reverse_iterator = /* implementation defined */;
7     using const_reverse_iterator = /* implementation defined */;
8     using difference_type = /* implementation defined */;
9
10    public:
11        // Constructors and destructors
12        constexpr path_view() noexcept;
13
14        path_view(path_view_component p, format fmt = path::auto_format) noexcept;
15
16        path_view(const path& p) noexcept;
17        template<class CharT>
18        constexpr path_view(const basic_string<CharT>,
19                            format fmt = path::auto_format) noexcept;
20
21        template<class CharT>
22        constexpr path_view(const CharT* b, size_type l, enum termination zt,
23                            format fmt = path::auto_format) noexcept;
24        constexpr path_view(const byte* b, size_type l, enum termination zt) noexcept;
25
26        template<class CharT>
27        constexpr path_view(const CharT* b, format fmt = path::auto_format) noexcept;
28        constexpr path_view(const byte* b) noexcept;
29
30        template<class CharT>
31        constexpr path_view(basic_string_view<CharT> b, enum termination zt,
32                            format fmt = path::auto_format) noexcept;
33        constexpr path_view(span<const byte> b, enum termination zt) noexcept;
34
35        template<class It, class End>
36        constexpr path_view(It b, End e, enum termination zt,
37                            format fmt = path::auto_format) noexcept;
38        template<class It, class End>
39        constexpr path_view(It b, End e, enum termination zt) noexcept;
40
41        constexpr path_view(const path_view&) = default;
42        constexpr path_view(path_view&&) = default;
43        constexpr ~path_view() = default;
44
45        // Assignments
46        constexpr path_view &operator=(const path_view&) = default;
47        constexpr path_view &operator=(path_view&&) = default;
48
49        // Modifiers
50        constexpr void swap(path_view& o) noexcept;
51
52        // Query
53        constexpr bool has_root_name() const noexcept;
54        constexpr bool has_root_directory() const noexcept;
55        constexpr bool has_root_path() const noexcept;
56        constexpr bool has_relative_path() const noexcept;
57        constexpr bool has_parent_path() const noexcept;
58        constexpr bool has_filename() const noexcept;
59        constexpr bool is_absolute() const noexcept;
60        constexpr bool is_relative() const noexcept;

```

```

61
62 constexpr path_view root_name() const noexcept;
63 constexpr path_view root_directory() const noexcept;
64 constexpr path_view root_path() const noexcept;
65 constexpr path_view relative_path() const noexcept;
66 constexpr path_view parent_path() const noexcept;
67 constexpr path_view_component filename() const noexcept;
68 constexpr path_view remove_filename() const noexcept;
69
70 // Iteration
71 constexpr const_iterator cbegin() const noexcept;
72 constexpr const_iterator begin() const noexcept;
73 constexpr iterator begin() noexcept;
74 constexpr const_iterator cend() const noexcept;
75 constexpr const_iterator end() const noexcept;
76 constexpr iterator end() noexcept;
77
78 // Comparison
79 template<class T = typename path::value_type,
80         class Allocator = default_rendered_path_allocator<T>,
81         size_type InternalBufferSize = default_internal_buffer_size>
82 constexpr int compare(path_view p) const;
83 template<class T = typename path::value_type,
84         class Allocator = default_rendered_path_allocator<T>,
85         size_type InternalBufferSize = default_internal_buffer_size>
86 constexpr int compare(path_view p, const locale &loc) const;
87
88 // Conversion
89 template<class T = typename path::value_type,
90         class Allocator = default_rendered_path_allocator<T>,
91         size_type InternalBufferSize = default_internal_buffer_size>
92 class rendered_path;
93 };
94
95 // Comparison
96 inline constexpr bool operator==(path_view a, path_view b) noexcept;
97 inline constexpr strong_ordering operator<=>(path_view a, path_view b) noexcept;
98
99 // Disabled comparisons
100 template<class CharT>
101 inline constexpr bool operator==(path_view, const CharT*) = delete(/* implementation defined message
102     */);
103 template<class CharT>
104 inline constexpr bool operator==(path_view, basic_string_view<CharT>) = delete(/* implementation
105     defined message */);
106 inline constexpr bool operator==(path_view, const byte*) = delete(/* implementation defined message
107     */);
108 inline constexpr bool operator==(path_view, span<const byte>) = delete(/* implementation defined
109     message */);
110
111 template<class CharT>
112 inline constexpr auto operator<=>(path_view, const CharT*) = delete(/* implementation defined
113     message */);
114 template<class CharT>
115 inline constexpr auto operator<=>(path_view, basic_string_view<CharT>) = delete(/* implementation
116     defined message */);

```

```

111 inline constexpr auto operator<=>(path_view, const byte*) = delete(/* implementation defined message
    */);
112 inline constexpr auto operator<=>(path_view, span<const byte>) = delete(/* implementation defined
    message */);
113
114 template<class CharT>
115 inline constexpr bool operator==(const CharT*, path_view) = delete(/* implementation defined message
    */);
116 template<class CharT>
117 inline constexpr bool operator==(basic_string_view<CharT>, path_view) = delete(/* implementation
    defined message */);
118 inline constexpr bool operator==(const byte*, path_view) = delete(/* implementation defined message
    */);
119 inline constexpr bool operator==(span<const byte>, path_view) = delete(/* implementation defined
    message */);
120
121 template<class CharT>
122 inline constexpr auto operator<=>(const CharT*, path_view) = delete(/* implementation defined
    message */);
123 template<class CharT>
124 inline constexpr auto operator<=>(basic_string_view<CharT>, path_view) = delete(/* implementation
    defined message */);
125 inline constexpr auto operator<=>(const byte*, path_view) = delete(/* implementation defined message
    */);
126 inline constexpr auto operator<=>(span<const byte>, path_view) = delete(/* implementation defined
    message */);
127 }

```

Path view iterators iterate over the elements of the path view as separated by the generic or native path separator, depending on the value of `formatting()`.

A `path_view::iterator` is a constant iterator meeting all the requirements of a bidirectional iterator. Its `value_type` is `path_view_component`.

Any operation that invalidates a pointer within the range of the backing data of the path view invalidates pointers, iterators and references returned by `path_view`.

For the elements of the pathname, the forward traversal order is as follows:

- The *root-name* element, if present.
- The *root-directory* element, if present.
- Each successive *filename* element, if present.
- An empty element, if a trailing non-root *directory-separator* is present.

The backward traversal order is the reverse of forward traversal. The iteration of any path view is required to be identical to the iteration of any path, for the same input path.

## Construction and assignment [`fs.path_view.cons`]

[*Note:* Apart from the default value for `format`, the path view constructors and assignment are identical to the path view component constructors, and are not repeated here for brevity. – end note]

## Observers [fs.path\_view.observers]

```
1 constexpr bool has_root_name() const noexcept;
```

*Returns:* True if `root_name()` returns a non-empty path view.

*Complexity:* `O(native_size())`.

```
1 constexpr bool has_root_directory() const noexcept;
```

*Returns:* True if `root_directory()` returns a non-empty path view.

*Complexity:* `O(native_size())`.

```
1 constexpr bool has_root_path() const noexcept;
```

*Returns:* True if `root_path()` returns a non-empty path view.

*Complexity:* `O(native_size())`.

```
1 constexpr bool has_relative_path() const noexcept;
```

*Returns:* True if `relative_path()` returns a non-empty path view.

*Complexity:* `O(native_size())`.

```
1 constexpr bool has_parent_path() const noexcept;
```

*Returns:* True if `parent_path()` returns a non-empty path view.

*Complexity:* `O(native_size())`.

```
1 constexpr bool has_filename() const noexcept;
```

*Returns:* True if `filename()` returns a non-empty path view component.

*Complexity:* `O(native_size())`.

```
1 constexpr bool is_absolute() const noexcept;
```

*Returns:* True if the path view contains an absolute path after interpretation by `formatting()`.



```
1 constexpr bool is_relative() const noexcept;
```

*Returns:* True if `is_absolute()` is false.

```
1 constexpr path_view root_name() const noexcept;
```

*Returns:* A path view referring to the subset of this path view if it contains *root-name*, otherwise an empty path view.

*Complexity:* `O(native_size())`.

```
1 constexpr path_view root_directory() const noexcept;
```

*Returns:* A path view referring to the subset of this path view if it contains *root-directory*, otherwise an empty path view.

*Complexity:* `O(native_size())`.

```
1 constexpr path_view root_path() const noexcept;
```

*Returns:* A path view referring to the subset of this path view if it contains *root-name sep root-directory* where *sep* is interpreted according to `formatting()`.

*Complexity:* `O(native_size())`.

```
1 constexpr path_view relative_path() const noexcept;
```

*Returns:* A path view referring to the subset of this view from the first filename after `root_path()` until the end of the view, which may be an empty view.

*Complexity:* `O(native_size())`.

```
1 constexpr path_view parent_path() const noexcept;
```

*Returns:* `*this` if `has_relative_path()` is false, otherwise a path view referring to the subset of this view from the beginning until the last *sep* exclusive, where *sep* is interpreted according to `formatting()`.

*Complexity:* `O(native_size())`.

```
1 constexpr path_view_component filename() const noexcept;
```

*Returns:* `*this` if `has_relative_path()` is false, otherwise `*--end()`.

*Complexity:* `O(native_size())`.

```
1 constexpr path_view remove_filename() const noexcept;
```

*Returns:* A path view referring to the subset of this view from the beginning until the last *sep* inclusive, where *sep* is interpreted according to `formatting()`.

*Complexity:* `O(native_size())`.

```
1 template<class T = typename path::value_type,
2         class Allocator = default_rendered_path_allocator<T>,
3         size_type InternalBufferSize = default_internal_buffer_size>
4 constexpr int compare(path_view p) const;
5 template<class T = typename path::value_type,
6         class Allocator = default_rendered_path_allocator<T>,
7         size_type InternalBufferSize = default_internal_buffer_size>
8 constexpr int compare(path_view p, const locale &loc) const;
```

*Returns:* Each path view is iterated from begin to end, and the path view components are compared. If any of those path view component comparisons return not zero, that value is returned. If the iteration sequence ends earlier for *\*this*, a negative number is returned; if the iteration sequence ends earlier for the externally supplied path view, a positive number is returned; if both iteration sequences have the same length, and all path component comparisons return zero, zero is returned.

*Complexity:* `O(native_size())`.

## Non-member comparison functions [fs.path\_view.comparison]

```
1 inline constexpr bool operator==(path_view a, path_view b) noexcept;
```

*Effects:* Iterates each `path_view_component` in both path views in sequence, if any compare false this returns false. If the number of path view components is not equal returns false. Otherwise returns true.

```
1 inline constexpr strong_ordering operator<=>(path_view a, path_view b) noexcept;
```

*Effects:* Iterates each `path_view_component` in both path views in sequence, if any compare non-equal returns that ordering. If the number of path view components is non-equal, returns that ordering. Otherwise returns equal.

```
1 template<class CharT>
2 inline constexpr bool operator==(path_view, const CharT*) = delete(/* implementation defined message
3 */);
4 template<class CharT>
5 inline constexpr bool operator==(path_view, basic_string_view<CharT>) = delete(/* implementation
6 defined message */);
7 inline constexpr bool operator==(path_view, const byte*) = delete(/* implementation defined message
8 */);
9 inline constexpr bool operator==(path_view, span<const byte>) = delete(/* implementation defined
10 message */);
```

```

7
8  template<class CharT>
9  inline constexpr auto operator<=>(path_view, const CharT*) = delete(/* implementation defined
10     message */);
11  template<class CharT>
12  inline constexpr auto operator<=>(path_view, basic_string_view<CharT>) = delete(/* implementation
13     defined message */);
14  inline constexpr auto operator<=>(path_view, const byte*) = delete(/* implementation defined message
15     */);
16  inline constexpr auto operator<=>(path_view, span<const byte>) = delete(/* implementation defined
17     message */);
18
19  template<class CharT>
20  inline constexpr bool operator==(const CharT*, path_view) = delete(/* implementation defined message
21     */);
22  template<class CharT>
23  inline constexpr bool operator==(basic_string_view<CharT>, path_view) = delete(/* implementation
24     defined message */);
25  inline constexpr bool operator==(const byte*, path_view) = delete(/* implementation defined message
26     */);
27  inline constexpr bool operator==(span<const byte>, path_view) = delete(/* implementation defined
28     message */);

```

*Effects:* Comparing string literals, string views, byte literals or byte views, against a path view is deleted. A diagnostic explaining that `.compare()` or `visit()` ought to be used instead is suggested.

### In 31.12.6.9.1 [fs.path.fmtr.general]:

```

+ template<class charT> struct formatter<filesystem::path_view_component, charT> {
+ constexpr void set_debug_format();

+ constexpr typename basic_format_parse_context<charT>::iterator
+ parse(basic_format_parse_context<charT>& ctx);

+ template<class FormatContext>
+ typename FormatContext::iterator
+ format(const filesystem::path_view_component& path_view, FormatContext& ctx) const;

```

```

+ };

+ template<class charT> struct formatter<filesystem::path_view, charT> {
+ constexpr void set_debug_format();

+ constexpr typename basic_format_parse_context<charT>::iterator
+ parse(basic_format_parse_context<charT>& ctx);

+ template<class FormatContext>
+ typename FormatContext::iterator
+ format(const filesystem::path_view& path_view, FormatContext& ctx) const;
+ };

```

### In 31.12.6.9.2 [fs.path.fmtr.funcs]:

Formatting of paths uses formatting specifiers of the form

*path-format-spec*:

*fill-and-alignopt widthopt ?opt gopt*

where the productions *fill-and-align* and *width* are described in 22.14.2. If the *?* option is used then the path is formatted as an escaped string (22.14.6.5).+ If the *??* option is used then the path is formatted as an escaped string (22.14.6.5) only if the backing representation of its view is `byte`.

### In 31.12.6.9.2 [fs.path.fmtr.funcs] paragraph 7:

```

1  template<class FormatContext>
2  typename FormatContext::iterator
3  format(const filesystem::path_view_component& p, FormatContext& ctx) const;
4
5  template<class FormatContext>
6  typename FormatContext::iterator
7  format(const filesystem::path_view& p, FormatContext& ctx) const;

```

*Effects*: Let *s* be as if `path(p).generic_string<filesystem::path::value_type>()` if the *g* option is used, otherwise `p.render_unterminated()`. Writes *s* into `ctx.out()`, adjusted according to the *path-format-spec*. If `charT` is `char`, `path::value_type` is `wchar_t`, and the literal encoding is UTF-8, then the escaped path is transcoded from the native encoding for wide character strings to UTF-8 with maximal subparts of ill-formed subsequences substituted with `u+ffff` replacement character per the Unicode Standard, Chapter 3.9 `u+ffff` Substitution in Conversion. If `charT` and `path::value_type` are the same then no transcoding is performed. Otherwise, transcoding is implementation-defined.

*Returns:* An iterator past the end of the output range.

### In 31.12.6.10 [fs.path.hash]:

```
+ template<> struct hash<filesystem::path_view_component>;  
+ template<> struct hash<filesystem::path_view>;
```

### In 31.12.8.1 [fs.enum.path.format] paragraph 1:

Name	Meaning
native_format	The native pathname format.
generic_format	The generic pathname format.
+ binary_format	+ The binary pathname format.
auto_format	The interpretation of the format of the character sequence is implementation-defined. The implementation may inspect the content of the character sequence to determine the format. <i>Recommended practice:</i> For POSIX-based systems, native and generic formats are equivalent and the character sequence should always be interpreted in the same way.

## 6 Acknowledgements

Too many to list! Thanks to you all.

## 7 References

[P0482] Tom Honermann,  
*char8\_t: A type for UTF-8 characters and strings*  
<https://wg21.link/P0482>

[P0882] Yonggang Li  
*User-defined Literals for std::filesystem::path*  
<https://wg21.link/P0882>

[P1031] Douglas, Niall  
*Low level file i/o library*  
<https://wg21.link/P1031>