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. 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.

Draft R6 of this paper was written after dinner up to 3am whilst at the Varna WG21 meeting for presentation the following day (which didn’t happen, LEWG ran out of time).

My two co-creators were Robert Leahy and Elias Kosunen, without whom this R6 would not have happened. I am very grateful.

Contents

1 Design goals .......................... 2
   1.1 path_view_component and path_view .................................. 2
   1.2 path_view::rendered_path ............................................. 4

2 Change tracking log for LWG since R4 5

3 Delta from N4861 6

4 Acknowledgements 44

5 References 44
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. 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\(^1\), 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.

\(^1\)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  \texttt{path\_view::rendered\_path}

- \texttt{render\_null\_terminated()} and \texttt{render\_unterminated()} returning a \texttt{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 \texttt{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 \texttt{rendered\_path}, dynamic memory allocation is usually avoided through the use of a reasonably large inline buffer. This makes \texttt{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 \texttt{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 \texttt{rendered\_path} is unwound in the usual way. Therefore the large size is not the problem it might otherwise be.

- \texttt{rendered\_path} is intended to be storable within STL containers as that can be useful sometimes, and provides assignment so a single stack allocated \texttt{rendered\_path} instance can be reused for multiple path view inputs during a function. Via template parameters, \texttt{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 \texttt{rendered\_path} to be as large as the maximum possible path (e.g. \texttt{PATH\_MAX}) and thus guarantee that no dynamic memory allocation can ever occur.

- For typical end users, \texttt{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 \texttt{allocator} instance as a parameter:

```cpp
namespace detail {
    struct thread_local_scratch_allocator_t {
        char *allocate(size_t);
        void deallocate(void *, size_t);
    }
}

std::filesystem::path_view v("foo");
auto rpath = v.render_null_terminated(detail::thread_local_scratch_allocator_t{});
int fd = ::open(rpath.c_str(), O_RDONLY);
```
2 Change tracking log for LWG since R4

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

- 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`

3 Delta from N4861

The following normative wording delta is against [https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/n4901.pdf](https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/n4901.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 17.3.2 [version.syn] paragraph 2:

```cpp
#define __cpp_lib_filesystem 202109L // in <filesystem>
```
In 29.10.2 [filebuf]:

template<class charT, class traits = char_traits<charT>>
class basic_filebuf : public basic_streambuf<charT, traits> {
    public:
    // ...  

    // [filebuf.members], members
    bool is_open() const;
    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; see [fstream.syn]
    basic_filebuf* open(const string& s,
                        ios_base::openmode mode);
    basic_filebuf* open(const filesystem::path& s,
                        ios_base::openmode mode);

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

    // ...
};

In 29.10.2.4 [filebuf.members] paragraph 7:

+ basic_filebuf* open(path-view-like s, ios_base::openmode mode);
+ Remarks: Behaves as if return open(filesystem::path(s), mode);

In 29.10.3 [ifstream]:

template<class charT, class traits = char_traits<charT>>
class basic_ifstream : public basic_streambuf<charT, traits> {
    public:
    // ...

    // [ifstream.cons], constructors
    basic_ifstream();
    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; see [fstream.syn]
    explicit basic_ifstream(const string& s,
                             ios_base::openmode mode = ios_base::in);

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

template<T>
    explicit basic_ifstream(const T& s, ios_base::openmode mode = ios_base::in);
In 29.10.3.1 [ifstream.cons] paragraph 4:

+ explicit basic_ifstream(path-view-like s, ios_base::openmode mode = ios_base::in);
+ Remarks: Behaves as if the arguments to the constructor were filesystem::path(s.view), mode

In 29.10.3.3 [ifstream.members] paragraph 4:

+ void open(path-view-like s, ios_base::openmode mode = ios_base::in);
+ Remarks: Behaves as if return open(filesystem::path(s.view), mode);

In 29.10.4 [ofstream]:

```cpp
template<class charT, class traits = char_traits<charT>>
class basic_ofstream : public basic_streambuf<charT, traits> {
  public:

  // ...

  // [ofstream.cons], constructors
  basic_ofstream();
  explicit basic_ofstream(const char* s,
           ios_base::openmode mode = ios_base::in);
```
explicit basic_ofstream(const filesystem::path::value_type* s,  
    ios_base::openmode mode = ios_base::in); // wide systems only; see [fstream.syn]
explicit basic_ofstream(const string& s,  
    ios_base::openmode mode = ios_base::in);

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

template<class T>
explicit basic_ofstream(const T& s, ios_base::openmode mode = ios_base::in);
basic_ofstream(const basic_ofstream&) = delete;
basic_ofstream(basic_ofstream&& rhs);

basic_ofstream& operator=(const basic_ofstream&) = delete;
basic_ofstream& operator=(basic_ofstream&& rhs);

// [ofstream.swap], swap
void swap(basic_ofstream& rhs);

// [ofstream.members], members
basic_filebuf<charT, traits>** rdbuf() const;

bool is_open() const;
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; see [fstream.syn]
void open(const string& s, ios_base::openmode mode = ios_base::out);
void open(const filesystem::path& s, ios_base::openmode mode = ios_base::out);

void close();

// ...
In 29.10.5 \([\text{fstream}]\):

```cpp
template<class charT, class traits = char_traits<charT>>
class basic_fstream : public basic_streambuf<charT, traits> {
  // ...

  // [fstream.cons], constructors
  basic_fstream();
  explicit basic_fstream(const char* s,
                         ios_base::openmode mode = ios_base::in);
  explicit basic_fstream(const filesystem::path::value_type* s,
                         ios_base::openmode mode = ios_base::in); // wide systems only; see [fstream.syn]
  explicit basic_fstream(const string& s,
                         ios_base::openmode mode = ios_base::in);
  template<class T>
  explicit basic_fstream(const T& s, ios_base::openmode mode = ios_base::in | ios_base::out);

  basic_fstream(const basic_fstream&) = delete;
  basic_fstream(basic_fstream&& rhs);

  basic_fstream& operator=(const basic_fstream&) = delete;
  basic_fstream& operator=(basic_fstream&& rhs);

  // [fstream.swap], swap
  void swap(basic_fstream& rhs);

  // [fstream.members], members
  basic_filebuf<charT, traits>* rdbuf() const;
  bool is_open() const;
  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; see [fstream.syn]
  void open(const string& s, ios_base::openmode mode = ios_base::in | ios_base::out);
  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);

  void close();

  // ...
}
```

In 29.10.5.2 \([\text{fstream.cons}]\) paragraph 3:

```cpp
+ explicit basic_fstream(path-view-like s, ios_base::openmode mode = ios_base::in | ios_base::out);
```
+ Remarks: Behaves as if the arguments to the constructor were filesystem::path(s.view), mode

In 29.10.5.3 [fstream.members] paragraph 4:

+ void open(path-view-like s, ios_base::openmode mode = ios_base::in | ios_base::out);
+ Remarks: Behaves as if return open(filesystem::path(s.view), mode);

In 29.12.4 [fs.filesystem.syn]:

namespace std::filesystem {
    class path;
    + 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& lhs, path& rhs);
    size_t hash_value(path p);
    size_t hash_value(path p);
    + void swap(path_view_component& lhs, path_view_component& rhs);
    + size_t hash_value(path_view_component p);
    + size_t hash_value(path_view p);
    // ...
    path absolute(const path& p);
    path absolute(const path& p, error_code& ec);
    + path absolute(path-view-like p);
    + path absolute(path-view-like p, error_code& ec);
    path canonical(const path& p);
    path canonical(const path& p, error_code& ec);
    + path canonical(path-view-like p);
    + path canonical(path-view-like p, error_code& ec);
    void copy(const path& from, const path& to);
    void copy(const path& from, const path& to, 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);
    void copy(const path& from, const path& to, copy_options options);
    void copy(const path& from, const path& to, copy_options options, error_code& ec);
void create_symlink(const path& to, const path& new_symlink);
void create_symlink(const path& to, const path& new_symlink, 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;

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

bool equivalent(const path& p1, const path& p2);
bool equivalent(const path& p1, const path& p2, 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(file_status s) noexcept;
bool exists(const path& p);
bool exists(const path& p, error_code& ec) noexcept;
+ bool exists(path-view-like p);
+ bool exists(path-view-like p, error_code& ec) noexcept;

uintmax_t file_size(const path& p);
uintmax_t file_size(const path& 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(const path& p);
uintmax_t hard_link_count(const path& 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(file_status p) noexcept;
bool is_block_file(const path& p);
bool is_block_file(const path& 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(file_status p) noexcept;
bool is_character_file(const path& p);
bool is_character_file(const path& 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(file_status p) noexcept;
bool is_directory(const path& p);
bool is_directory(const path& p, error_code& ec) noexcept;

bool is_directory(file_status s) noexcept;
bool is_directory(const path& p);
bool is_directory(const path& 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(file_status p)noexcept;
bool is_empty(const path& p);
bool is_empty(const path& 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(file_status p)noexcept;
bool is_fifo(const path& p);
bool is_fifo(const path& 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(file_status p)noexcept;
bool is_other(const path& p);
bool is_other(const path& 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(file_status p)noexcept;
bool is_regular_file(const path& p);
bool is_regular_file(const path& 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(file_status p)noexcept;
bool is_socket(const path& p);
bool is_socket(const path& 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(file_status p)noexcept;
bool is_symlink(const path& p);
bool is_symlink(const path& 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(const path& p);
file_time_type last_write_time(const path& 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(const path& p, file_time_type new_time);
void last_write_time(const path& p, file_time_type new_time, 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(const path& p, perms prms, perm_options opts=perm_options::replace);
void permissions(const path& p, perms prms, error_code& ec) noexcept;
void permissions(const path& p, perms prms, perm_options opts, error_code& ec);
+ 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(const path& p, error_code& ec);
path proximate(const path& p, const path& base = current_path());
path proximate(const path& p, perm_options opts=perm_options::replace);
path proximate(const path& p, const path& base, 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(const path& p);
path read_symlink(const path& p, error_code& ec);
+ path read_symlink(path-view-like p);
+ path read_symlink(path-view-like p, error_code& ec);

path relative(const path& p, error_code& ec);
path relative(const path& p, const path& base = current_path());
path relative(const path& p, perm_options opts=perm_options::replace);
path relative(const path& p, const path& base, 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(const path& p);
bool remove(const path& p, error_code& ec) noexcept;
+ bool remove(path-view-like p);
+ bool remove(path-view-like p, error_code& ec) noexcept;

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

void rename(const path& from, const path& to);
void rename(const path& from, const path& to, 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(const path& p, uintmax_t size);
void resize_file(const path& p, uintmax_t size, 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(const path& p);
space_info space(const path& p, error_code& ec) noexcept;
+ space_info space(path-view-like p);
+ space_info space(path-view-like p, error_code& ec) noexcept;
```cpp
namespace std {
  template<class T> struct hash;
  template<> struct hash<filesystem::path>;
  template<> struct hash<filesystem::path_view_component>;
  template<> struct hash<filesystem::path_view>;
}
```

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.
In 29.12.6 [fs.class.path.general] paragraph 6:

```cpp
class path {
public:
    using value_type = see below;
    using string_type = basic_string<value_type>;
    static constexpr value_type preferred_separator = see below;

    // [fs.enum.path.format], enumeration format
    enum format;

    // [fs.path.construct], constructors and destructor
    path() noexcept;
    path(const path& p);
    path(path&& p) noexcept;
    explicit path(path-view-like p);
    path(string_type&& source, format fmt = auto_format);
    template<class Source>
    path(const Source& source, format fmt = auto_format);
    template<class InputIterator>
    path(InputIterator first, InputIterator last, format fmt = auto_format);
    template<class Source>
    path(const Source& source, const locale& loc, format fmt = auto_format);
    template<class InputIterator>
    path(InputIterator first, InputIterator last, const locale& loc, format fmt = auto_format);
    ~path();

    // [fs.path.assign], assignments
    path& operator=(const path& p);
    path& operator=(path&& p) noexcept;
    path& assign(string_type&& source);
    template<class Source>
    path& operator=(const Source& source);
    template<class InputIterator>
    path& assign(InputIterator first, InputIterator last);

    // [fs.path.append], appends
    path& operator/=(const path& p);
    template<class Source>
    path& operator/=(const Source& source);
    template<class InputIterator>
    path& append(InputIterator first, InputIterator last);
```

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`.
path& append(InputIterator first, InputIterator last);

// [fs.path.concat], concatenation
path& operator+=(const path& x);
path& operator+=(const string_type& x);
path& operator+=(basic_string_view<value_type> x);
path& operator+=(const value_type* x);
path& operator+=(value_type x);

template<class Source>
path& operator+=(const Source& x);

path& operator+=(EcharT x);

path& concat(const Source& x);
path& concat(InputIterator first, InputIterator last);

// [fs.path.modifiers], modifiers
void clear() noexcept;
path& make_preferred();
path& remove_filename();
path& replace_filename(const path& replacement);

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

+ path& replace_extension(const path& replacement = path());

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

void swap(path& rhs) noexcept;

// [fs.path.nonmember], non-member operators
friend bool operator==(const path& lhs, const path& rhs) noexcept;
frend strong_ordering operator<(const path& lhs, const path& rhs) noexcept;
frend 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);

// [fs.path.native.obs], native format observers
const string_type& native() const +& noexcept;

+ string_type&& native()&& noexcept;

cost value_type* c_str() const noexcept;
operator string_type() const;

template<class EcharT, class traits = char_traits<EcharT>,
class Allocator = allocator<EcharT>>
basic_string<EcharT, traits, Allocator>

string(const Allocator& a = Allocator()) const;
std::string string() const;
std::wstring wstring() const;
std::u8string u8string() const;
std::u16string u16string() const;
std::u32string u32string() const;

// [fs.path.generic.obs], generic format observers
template<class EcharT, class traits = char_traits<EcharT>,
class Allocator = allocator<EcharT>>
basic_string<EcharT, traits, Allocator>
generic_string(const Allocator& a = Allocator()) const;
std::string generic_string() const;
std::wstring generic_wstring() const;
std::u8string generic_u8string() const;
std::u16string generic_u16string() const;
std::u32string generic_u32string() const;

// [fs.path.compare], compare
int compare(const path& p) const noexcept;
+ int compare(path-view-like p) const;
int compare(const string_type& s) const;
int compare(basic_string_view<value_type> s) const;
int compare(const value_type* s) const;

// [fs.path.decompose], decomposition
path root_name() const;
path root_directory() const;
path root_path() const;
path relative_path() const;
path parent_path() const;
path filename() const;
path stem() const;
path extension() const;

+ format formatting() const noexcept;

// [fs.path.query], query
[[nodiscard]] bool empty() const noexcept;
bool has_root_name() const;
bool has_root_directory() const;
bool has_root_path() const;
bool has_relative_path() const;
bool has_parent_path() const;
bool has_filename() const;
bool has_stem() const;
bool has_extension() const;
bool is_absolute() const;
bool is_relative() const;

// [fs.path.gen], generation
path lexically_normal() const;
path lexically_relative(const path& base) const;
+ path lexically_relative(path-view-like p)const;

path lexically_proximate(const path& base) const;

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

// [fs.path.itr], iterators
class iterator;
using const_iterator = iterator;

iterator begin() const;
iterator end() const;

// [fs.path.io], path inserter and extractor
template<class charT, class traits>
friend basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, const path& p);

template<class charT, class traits>
friend basic_istream<charT, traits>&
operator>>(basic_istream<charT, traits>& is, path& p);

};

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 29.12.6.5.1 [fs.path.construct]:

+ explicit path(path-view-like p);
+ Effects: Constructs an object of class path by an equivalent call to:

visit([&p](auto sv) -> path {
  if constexpr(same_as<remove_cvref_t<decltype(sv)>, span<const byte>>)
  {
    return path-from-binary(sv);
  }
  else
  {
    return path(sv, p.formatting());
  }
}, p.view);

+ format formatting() const noexcept;
+ Returns: The appropriate path separator format interpretation for the current path’s contents.
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.
constexpr path_view_component(const byte* b) noexcept;

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

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

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

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

constexpr path_view_component(const path_view_component&) = default;
constexpr path_view_component(path_view_component&&) = default;
constexpr ~path_view_component() = default;

// Assignments
constexpr path_view_component &operator=(const path_view_component&) = default;
constexpr path_view_component &operator=(path_view_component&&) = default;

// Modifiers
constexpr void swap(path_view_component& o) noexcept;

// Query
[[nodiscard]] constexpr bool empty() const noexcept;
constexpr size_type native_size() const noexcept;
constexpr format formatting() const noexcept;
constexpr bool has_null_termination() const noexcept;
constexpr enum termination termination() const noexcept;
constexpr bool has_stem() const noexcept;
constexpr bool has_extension() const noexcept;

constexpr path_view_component stem() const noexcept;
constexpr path_view_component extension() const noexcept;

// Comparison

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

// Conversion

template<
    enum path_view_component::termination Termination,
    class T = typename path::value_type,
    class Allocator = default_rendered_path_allocator<T>,
    size_type InternalBufferSize = default_internal_buffer_size>
class rendered_path;

// Conversion convenience

template <class T = typename path::value_type,
    class Allocator = default_rendered_path_allocator<T>,
    size_type InternalBufferSize = default_internal_buffer_size>
constexpr rendered_path<termination::null_terminated,
    T, Allocator, _internal_buffer_size>
render_null_terminated(Allocator allocate = Allocator());
template <class T = typename path::value_type,
    class Allocator = default_rendered_path_allocator<T>,
    size_type InternalBufferSize = default_internal_buffer_size>
constexpr rendered_path<termination::unterminated,
    T, Allocator, _internal_buffer_size>
render_unterminated(Allocator allocate = Allocator());

private:
{
    const byte* byestr_{nullptr}; // exposition only
    const char* charstr_; // exposition only
    const wchar_t* wcharstr_; // exposition only
    const char8_t* char8str_; // exposition only
    const char16_t* char16str_; // exposition only
    size_type length_{0}; // exposition only
    uint16_t null_terminated_ : 1; // exposition only
    uint16_t is_byestr_ : 1; // exposition only
    uint16_t is_charstr_ : 1; // exposition only
    uint16_t is_wcharstr_ : 1; // exposition only
    uint16_t is_char8str_ : 1; // exposition only
    uint16_t is_char16str_ : 1; // exposition only
    format format_{format::unknown}; // exposition only
};

/* Note to be removed before LWG: if your platform has a maximum path size
which fits inside a uint32_t, it is possible to pack path views
into 2 * sizeof(void*), which can be returned in CPU registers on
x64 Itanium ABI.
*/
static_assert(std::is_trivially_copyable_v<path_view_component>); // to be removed before LWG
static_assert(sizeof(path_view_component) == 2 * sizeof(void*)); // to be removed before LWG

// Comparison
inline constexpr bool operator==(path_view_component a, path_view_component b) noexcept;
inline constexpr bool operator<(path_view_component a, path_view_component b) noexcept;
inline constexpr auto operator<=>(path_view_component a, path_view_component b) = default;

// Disabled comparisons
template<class CharT>
inline constexpr bool operator==(path_view_component, const CharT*) = delete;
template<class CharT>
inline constexpr bool operator==(path_view_component, basic_string_view<CharT>) = delete;
inline constexpr bool operator==(path_view_component, const byte*) = delete;
inline constexpr bool operator==(path_view_component, span<const byte>) = delete;
template<class CharT>
inline constexpr bool operator<(path_view_component, const CharT*) = delete;
template<class CharT>
inline constexpr bool operator<(path_view_component, basic_string_view<CharT>) = delete;
inline constexpr bool operator<(path_view_component, const byte*) = delete;
inline constexpr bool operator<(path_view_component, span<const byte>) = delete;
template<class CharT>
inline constexpr auto operator<=>(path_view_component, const CharT*) = delete;
inline constexpr auto operator<=>(path_view_component, basic_string_view<CharT>) = delete;
inline constexpr auto operator<=>(path_view_component, const byte*) = delete;
inline constexpr auto operator<=>(path_view_component, span<const byte>) = delete;

template<class CharT>
inline constexpr bool operator==(const CharT*, path_view_component) = delete;
template<class CharT>
inline constexpr bool operator==(basic_string_view<CharT>, path_view_component) = delete;
inline constexpr bool operator==(const byte*, path_view_component) = delete;
inline constexpr bool operator==(span<const byte>, path_view_component) = delete;

template<class CharT>
inline constexpr bool operator<(const CharT*, path_view_component) = delete;
template<class CharT>
inline constexpr bool operator<(basic_string_view<CharT>, path_view_component) = delete;
inline constexpr bool operator<(const byte*, path_view_component) = delete;
inline constexpr bool operator<(span<const byte>, path_view_component) = delete;

// Hash value
size_t hash_value(path_view_component v) noexcept;

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

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

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\textsuperscript{2}.

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.

\textsuperscript{2}After much deliberation, LEWG chose 1,024 codepoints as a reasonable suggested default for most platforms.
• **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`.

2. `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`.

3. `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`.

4. ```template<class CharT> constexpr path_view_component(const basic_string<CharT>& s, 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`.  

25
template<class CharT>
constexpr path_view_component(const CharT* b, size_type l, enum termination zt,
    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.

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.

template<class CharT>
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)).

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.

整改：

```cpp
template<class CharT>
constexpr path_view_component(basic_string_view<CharT> b, enum termination zt,
    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`.

```cpp
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`.

```cpp
template<class It, class End>
constexpr path_view_component(It b, End e, enum termination zt,
    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`.

```cpp
template<class It, class End>
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]

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

Effects: Exchanges the values of *this and o.

Observers [fs.path_view_component.observers]

```cpp
[[nodiscard]] constexpr bool empty() const noexcept;
```

Returns: True if native_size()== 0.

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

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

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

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

```cpp
constexpr bool has_null_termination() const noexcept;
```
Returns: True if the path view component was constructed with zero termination.

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

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

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

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

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

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

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

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

```cpp
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.

```cpp
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: \(O(native\_size())\).

```cpp
template<class T = typename path::value_type,
         class Allocator = default_rendered_path_allocator<T>,
         size_type InternalBufferSize = default_internal_buffer_size>
constexpr int compare(path_view_component p) const;
```
**Constraints:** \( T \) is any one of: \( \text{char}, \text{wchar}_\text{t}, \text{char8}_\text{t}, \text{char16}_\text{t}, \text{byte} \); \( \text{Allocator} \) is either its defaulted internal tag type, or meets \( \text{Cpp17Allocator} \) requirements.

**Effects:**

- If \( T \) is \( \text{byte} \), the comparison of the two backing data ranges is implemented as a byte comparison equivalent to \( \text{memcmp} \).
- Otherwise the comparison is equivalent to:

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

**Complexity:** \( O(\text{native\_size}) \).

**Remarks:** The above wording is intended to retain an important source of optimisation whereby implementations do not actually have to construct a \( \text{path\_view\_component::rendered\_path} \) nor a \( \text{path} \) from those buffers e.g. if the backing data for both \( *\text{this} \) 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 \( \text{path} \) and those paths compared.

```

```

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

```

```

**Returns:** \( \text{rendered\_path<termination::unterminated, T, Allocator, _internal\_buffer\_size>>(*this, allocate)} \);

**Class** \( \text{path\_view\_component::rendered\_path} \) [fs.path\_view\_component.rendered\_path]

```

```
class Allocator = path_view_component::default_rendered_path_allocator<T>,
size_type InternalBufferSize = path_view_component::default_internal_buffer_size>
class path_view_component::rendered_path {
public:
using value_type = const T;
using pointer = const T*;
using const_pointer = const T*;
using reference = const T&;
using const_reference = const T&;
using iterator = span<value_type>::iterator;
using const_iterator = span<value_type>::const_iterator;
using reverse_iterator = span<value_type>::reverse_iterator;
using const_reverse_iterator = span<value_type>::const_reverse_iterator;
using size_type = span<value_type>::size_type;
using difference_type = span<value_type>::difference_type;
using allocator_type = Allocator; /* not present if default_rendered_path_allocator tag type was
used */

public:
// constructors and destructor
rendered_path() noexcept;
~rendered_path();

constexpr rendered_path(path_view_component v, Allocator allocate = Allocator());
rendered_path(const rendered_path&) = delete;
rendered_path(rendered_path&& o) noexcept;

// assignment
rendered_path &operator=(const rendered_path&) = delete;
rendered_path &operator=(rendered_path&&) noexcept;

// iteration
constexpr iterator begin() noexcept;
constexpr const_iterator begin() const noexcept;
constexpr const_iterator cbegin() const noexcept;
constexpr iterator end() noexcept;
constexpr const_iterator end() const noexcept;
constexpr const_iterator cend() const noexcept;
constexpr reverse_iterator rbegin() noexcept;
constexpr const_reverse_iterator rbegin() const noexcept;
constexpr const_reverse_iterator crbegin() const noexcept;
constexpr reverse_iterator rend() noexcept;
constexpr const_reverse_iterator rend() const noexcept;
constexpr const_reverse_iterator crend() const noexcept;

// access
constexpr reference operator[](size_type idx) noexcept;
constexpr const_reference operator[](size_type idx) const noexcept;
constexpr reference at(size_type idx);
constexpr const_reference at(size_type idx) const;
constexpr reference front() noexcept;
constexpr const_reference front() const noexcept;
constexpr reference back() noexcept;
constexpr reference back() const noexcept;
constexpr pointer data() noexcept;
constexpr const_pointer data() noexcept;
constexpr size_type size() const noexcept;
constexpr size_type length() const noexcept;
constexpr size_type max_size() const noexcept;
[[nodiscard]] constexpr bool empty() noexcept;

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

constexpr size_t capacity() const noexcept;
constexpr bool references_source() const noexcept;

constexpr span<const value_type> as_span() const noexcept; // available only if null_terminated and non-byte backing

private:
    span<const value_type> _ref; // exposition only
    size_t bytes_to_delete_{0}; // exposition only
    Allocator allocator_; // exposition only
    value_type buffer_[internal_buffer_size]{}; // exposition only

    /* To be removed before LWG:
    */

Note that if the internal buffer is the final item in the structure,
the major C++ compilers shall, if they can statically prove that
the buffer will never be used, entirely eliminate it from runtime
codegen. This can happen quite frequently during aggressive
inlining if the backing data is a string literal.
*/
};

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.

As an example of usage with POSIX open(), which consumes a zero-terminated const path::value_type* i.e. const char *:

```cpp
int open_file(path_view path)
{
    /* This function does not support binary key input */
    if(visit([](auto sv){ return same_as<remove_cvref_t<decltype(sv)>, span<const byte>>; }, path))
    {
        errno = EINVAL;
        return -1;
    }
    /* On POSIX platforms which treat char as UTF-8, if the input has backing data in char or char8_t, and that backing data is zero terminated, zpath.data() will point into the backing data and no further work is done. Otherwise a reencode or bit copy of the backing data to char will be performed, possibly dynamically allocating a buffer if rendered.path's internal buffer isn't big enough. */
    auto zpath = path.render_null_terminated();
    return ::open(zpath.c_str(), O_RDONLY);
}
```

### Construction [fs.path_view_component.rendered_path.cons]

```cpp
~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.

```cpp
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 \texttt{v.formatting()} the backing data contains any generic path separators and the generic path separator is not the native path separator:

- If \texttt{size() < internal.buffer.size - 1}:
  
  * \texttt{data()} returns \texttt{buffer_*}, the code points of the backing data are copied into \texttt{buffer_*}, replacing any generic path separators with native path separators if \texttt{v.formatting()} allows that, and a zero valued \texttt{value_type} is appended.

  else:

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

  else:

  - \texttt{data()} returns the backing data.

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

### Observers [fs.rendered_path.obs]

\textit{Note:} The vast majority of the observers replicate those of \texttt{span} and so are not described further here. The reason \texttt{span} was chosen over \texttt{basic_string_view} is because the rendered path could be binary. – end note

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

Constraints: \texttt{Allocator meets Cpp17Allocator} requirements.

\textit{Returns:} The allocator associated with the rendered path.
```

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

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

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

\textit{Returns:} True if this rendered path references backing data elsewhere.
```
constexpr span<const value_type> as_span() const noexcept;

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

constexpr const_pointer c_str() const noexcept; // available only if null_terminated and non-byte backing

**Constraints:** T is any one of: char, wchar_t, char8_t, char16_t; Termination is termination::null_terminated.

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

**Non-member comparison functions** [fs.path_view_component.comparison]

```cpp
inline constexpr bool operator==(path_view_component a, path_view_component b) noexcept;
inline constexpr bool operator<(path_view_component a, path_view_component b) noexcept;
inline constexpr auto operator<=>(path_view_component a, path_view_component b) = default;
```

**Effects:** If the native sizes are unequal, the path view components are considered unequal. If the backing bytes are of different encoding, the path view components are considered unequal. Otherwise a comparison equivalent to memcmp is used to compare the backing bytes of both path view components for equality and ordering.

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

```cpp
template<class CharT>
inline constexpr bool operator==(path_view_component, const CharT*) = delete;
template<class CharT>
inline constexpr bool operator==(path_view_component, basic_string_view<CharT>) = delete;
template<class CharT>
inline constexpr bool operator==(path_view_component, const byte*) = delete;
template<class CharT>
inline constexpr bool operator==(path_view_component, span<const byte>) = delete;
```

```cpp
template<class CharT>
inline constexpr bool operator<(path_view_component, const CharT*) = delete;
template<class CharT>
inline constexpr bool operator<(path_view_component, basic_string_view<CharT>) = delete;
template<class CharT>
inline constexpr bool operator<(path_view_component, const byte*) = delete;
template<class CharT>
inline constexpr bool operator<(path_view_component, span<const byte>) = delete;
```

```cpp
template<class CharT>
inline constexpr auto operator<=>(path_view_component, const CharT*) = delete;
template<class CharT>
inline constexpr auto operator<=>(path_view_component, basic_string_view<CharT>) = delete;
template<class CharT>
inline constexpr auto operator<=>(path_view_component, const byte*) = delete;
template<class CharT>
inline constexpr auto operator<=>(path_view_component, span<const byte>) = delete;
```

```cpp
template<class CharT>
inline constexpr bool operator==(const CharT*, path_view_component) = delete;
```
Effects: Comparing for equality or inequality 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 recommended.

Non-member functions [fs.path_view_component.comparison]

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).

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

**Effects:** Equivalent to:

```cpp
template<class charT, class traits>
basic_ostream<charT, traits>& operator<<(basic_ostream<charT, traits>& os, path_view_component v)
{
    return visit([&os, &v](auto sv) -> basic_ostream<charT, traits>& {
        using input_type = remove_cvref_t<decltype(sv)>
        using output_type = basic_ostream<charT, traits>;

        if constexpr(same_as<input_type, span<const byte>>)
        {
            /* Handle byte encoded filesystem paths however the
            implementation handles them. For example, Microsoft Windows
            requires the following textualisation for
            FILE_ID_DESCRIPTOR.ObjectId keys which are guids:
            
            "{7ec65a0-4b78-5f9b-e77c-8770091c0100}"
            
            This is a valid filename in NTFS with special semantics:
            OpenFileById() is used instead if you pass it into
            CreateFile().
            
            Otherwise some textual representation which is not
            a possible valid textual path is suggested.
            */
            return os << quoted(\emph{path-from-binary}(sv).string<typename output_type::char_type>());
        }
        else
        {
            // Possibly reencode to ostream’s character type
            path_view_component::rendered_path<typename output_type::char_type> zbuff(v, path_view_component::unterminated);
            return os << quoted(basic_string_view<typename output_type::char_type>(zbuff.buffer, zbuff.length));
        }
    }, v);
}
```

**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.
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 \texttt{path\_view}.

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

- The \texttt{root-name} element, if present.
- The \texttt{root-directory} element, if present.
- Each successive \texttt{filename} element, if present.
- An empty element, if a trailing non-root \texttt{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.

\textbf{Construction and assignment [fs.path\_view.cons]}

\begin{quote}
\textit{Note:} Apart from the default value for \texttt{format}, the path view constructors and assignment are identical to the path view component constructors, and are not repeated here for brevity. – end note
\end{quote}

\textbf{Observers [fs.path\_view.observers]}

\begin{verbatim}
1 constexpr bool has_root_name() const noexcept;
\end{verbatim}

\textit{Returns:} True if \texttt{root\_name()} returns a non-empty path view.

\textit{Complexity:} \texttt{O(native\_size())}.

\begin{verbatim}
1 constexpr bool has_root_directory() const noexcept;
\end{verbatim}

\textit{Returns:} True if \texttt{root\_directory()} returns a non-empty path view.

\textit{Complexity:} \texttt{O(native\_size())}.

\begin{verbatim}
1 constexpr bool has_root_path() const noexcept;
\end{verbatim}

\textit{Returns:} True if \texttt{root\_path()} returns a non-empty path view.

\textit{Complexity:} \texttt{O(native\_size())}.

\begin{verbatim}
1 constexpr bool has_relative_path() const noexcept;
\end{verbatim}

\textit{Returns:} True if \texttt{relative\_path()} returns a non-empty path view.

\textit{Complexity:} \texttt{O(native\_size())}.
constexpr bool has_parent_path() const noexcept;

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

Complexity: O(native_size()).

constexpr bool has_filename() const noexcept;

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

Complexity: O(native_size()).

constexpr bool is_absolute() const noexcept;

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

constexpr bool is_relative() const noexcept;

Returns: True if is_absolute() is false.

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()).

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()).

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()).

constexpr path_view relative_path() const noexcept;
Returns: A path view referring to the subset of this view from the first filename after \texttt{root.path()} until the end of the view, which may be an empty view.

\textit{Complexity:} \texttt{O(native.size())}.

\begin{verbatim}
constexpr path_view parent_path() const noexcept;
\end{verbatim}

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

\textit{Complexity:} \texttt{O(native.size())}.

\begin{verbatim}
constexpr path_view::component filename() const noexcept;
\end{verbatim}

\textit{Returns:} \texttt{*this} if \texttt{has_relative_path()} is false, otherwise \texttt{*--end()}.

\textit{Complexity:} \texttt{O(native.size())}.

\begin{verbatim}
constexpr path_view remove_filename() const noexcept;
\end{verbatim}

\textit{Returns:} A path view referring to the subset of this view from the beginning until the last \texttt{sep} inclusive, where \texttt{sep} is interpreted according to \texttt{formatting()}.

\textit{Complexity:} \texttt{O(native.size())}.

\begin{verbatim}
template<class T = typename path::value_type,
    class Allocator = default_rendered_path_allocator<T>,
    size_type InternalBufferSize = default_internal_buffer_size>
constexpr int compare(path_view p) const;
template<class T = typename path::value_type,
    class Allocator = default_rendered_path_allocator<T>,
    size_type InternalBufferSize = default_internal_buffer_size>
constexpr int compare(path_view p, const locale &loc) const;
\end{verbatim}

\textit{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 \texttt{*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.

\textit{Complexity:} \texttt{O(native.size())}.

\textbf{In 29.12.8.1 [fs.enum.path.format] paragraph 1:}

\begin{verbatim}
template<class T = typename path::value_type,
    class Allocator = default_rendered_path_allocator<T>,
    size_type InternalBufferSize = default_internal_buffer_size>
constexpr int compare(path_view p, const locale &loc) const;
\end{verbatim}
<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>native_format</td>
<td>The native pathname format.</td>
</tr>
<tr>
<td>generic_format</td>
<td>The generic pathname format.</td>
</tr>
<tr>
<td>+ binary_format</td>
<td>+ The binary pathname format.</td>
</tr>
<tr>
<td>auto_format</td>
<td>The interpretation of the format of the character sequence is</td>
</tr>
<tr>
<td></td>
<td>implementation-defined. The implementation may inspect the content of</td>
</tr>
<tr>
<td></td>
<td>the character sequence to determine the format. <strong>Recommended practice:</strong></td>
</tr>
<tr>
<td></td>
<td>For POSIX-based systems, native and generic formats are equivalent and</td>
</tr>
<tr>
<td></td>
<td>the character sequence should always be interpreted in the same way.</td>
</tr>
</tbody>
</table>

## 4 Acknowledgements

Draft R6 of this paper was written after dinner up to 3am whilst at the Varna WG21 meeting for presentation the following day (which didn’t happen, LEWG ran out of time).

My two cocreators were Robert Leahy and Elias Kosunen, without whom this R6 would not have happened. I am very grateful.

## 5 References

[P0482] Tom Honermann,  
*char8_t: A type for UTF-8 characters and strings*  
[https://wg21.link/P0482](https://wg21.link/P0482)

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

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