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.

My apologies for the two years which have elapsed since R4. ‘Real life’ has intervened on my standards paper writing, and it is only due to my current client MayStreet London Stock Exchange Group allowing me a few work hours to work on standards papers that you see this R5 now and not yet more months from now. I hope that normal service will resume in 2023. I will not be at the upcoming Kona meeting, but I currently expect to be at the New York meeting in early 2023.
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() will render a path view to a destination encoding and zero 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{path\_view::render()} 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 \texttt{v}("foo");
auto \texttt{rpath} = \texttt{v.render\_zero\_terminated}(detail::thread\_local\_scratch\_allocator\_t{});
int \texttt{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.

- 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.

3 Delta from N4861

The following normative wording delta is against 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 16.4.4.7 PathView and NonLegacyPath requirements [path_view.requirements]
[After 16.4.4.6 Cpp17Allocator requirements]:

In 16.4.4.7.1 General [path_view.requirements.general]:

+ The library contains many functions which accept a filesystem path. For backwards compatibility, overload sets from previous standard revisions ought to be selected preferentially to path view overloads. The PathView and NonLegacyPath requirements enable overloads for non-legacy path inputs.
In 16.4.4.7.2 *PathView* requirement [path_view.requirements.pathview]:

+ A type \( P \) meets the *PathView* requirements if it meets any one of:
  
  - It is a `filesystem::path_view_component`, or reference to that.
  - It is a `filesystem::path_view`, or reference to that.

In 16.4.4.7.3 *NonLegacyPath* requirement [path_view.requirements.nonlegacypath]:

+ A type \( P \) meets the *NonLegacyPath* requirements if it meets any one of:
  
  - It is a `filesystem::path_view_component`, or reference to that.
  - It is a `filesystem::path_view`, or reference to that.
  - It is convertible to `const byte*`.
  - It is convertible to `span<const byte>`.

In 17.3.2 [version.syn] paragraph 2:

```cpp
#define __cpp_lib_filesystem 201703L 202209L // also in <filesystem>
```

In 29.10.2 [filebuf]:

```cpp
+ template<NonLegacyPath T>
+ basic_filebuf* open(T&& s, ios_base::openmode mode);
```

In 29.10.2.4 [filebuf.members] paragraph 7:

```cpp
+ template<NonLegacyPath T>
+ basic_filebuf* open(T&& s, ios_base::openmode mode);
+ Effects: As if open(render_zero_terminated(path_view(forward<T&&>(s))).c_str(), mode);
```

In 29.10.3 [ifstream]:

```cpp
+ template<NonLegacyPath T>
+ explicit basic_ifstream(T&& s, ios_base::openmode mode = ios_base::in);
+ template<NonLegacyPath T>
+ void open(T&& s, ios_base::openmode mode = ios_base::in);
```
In 29.10.3.1 [ifstream.cons] paragraph 4:

+ template<NonLegacyPath T>
+ explicit basic_ifstream(T&& s, ios_base::openmode mode = ios_base::in);
+ Effects: As if basic_ifstream(render_zero_terminated(path_view(forward<T&&>(s))).c_str(), mode);

In 29.10.3.3 [ifstream.members] paragraph 4:

+ template<NonLegacyPath T>
+ void open(T&& s, ios_base::openmode mode = ios_base::in);
+ Effects: As if open(render_zero_terminated(path_view(forward<T&&>(s))).c_str(), mode);

In 29.10.4 [ofstream]:

+ template<NonLegacyPath T>
+ explicit basic ofstream(T&& s, ios_base::openmode mode = ios_base::out);
+ template<NonLegacyPath T>
+ void open(T&& s, ios_base::openmode mode = ios_base::out);

In 29.10.4.2 [ofstream.cons] paragraph 5:

+ template<NonLegacyPath T>
+ explicit basic ofstream(T&& s, ios_base::openmode mode = ios_base::out);
+ Effects: As if basic ofstream(render_zero_terminated(path_view(forward<T&&>(s))).c_str(), mode);

In 29.10.4.4 [ofstream.members] paragraph 3:

+ template<NonLegacyPath T>
+ void open(T&& s, ios_base::openmode mode = ios_base::out);
+ Effects: As if open(render_zero_terminated(path_view(forward<T&&>(s))).c_str(), mode);

In 29.10.5 [fstream]:

+ template<NonLegacyPath T>
+ explicit basic fstream(T&& s, ios_base::openmode mode = ios_base::in | ios_base::out);
+ template<NonLegacyPath T>
+ void open(T&& s, ios_base::openmode mode = ios_base::in | ios_base::out);
In 29.10.5.2 [fstream.cons] paragraph 3:

```cpp
+ template<NonLegacyPath T>
+ explicit basic_fstream(T&& s, ios_base::openmode mode = ios_base::in | ios_base::out);
+ Effects: As if basic_fstream(render_zero_terminated(path_view(forward<T&&>(s))).c_str(), mode);
```

In 29.10.5.3 [fstream.members] paragraph 4:

```cpp
+ template<NonLegacyPath T>
+ void open(T&& s, ios_base::openmode mode = ios_base::in | ios_base::out);
+ Effects: As if open(render_zero_terminated(path_view(forward<T&&>(s))).c_str(), mode);
```

In 29.12.4 [fs.filesystem.syn]:
+ template<NonLegacyPath T, NonLegacyPath U>
+ bool copy_file(T&& from, T&& to, error_code& ec);

+ template<NonLegacyPath T, NonLegacyPath U>
+ bool copy_file(T&& from, T&& to, copy_options options);

+ template<NonLegacyPath T, NonLegacyPath U>
+ bool copy_file(T&& from, T&& to, copy_options options, error_code& ec);

+ template<NonLegacyPath T, NonLegacyPath U>
+ void copy_symlink(T&& existing_symlink, T&& new_symlink);

+ template<NonLegacyPath T, NonLegacyPath U>
+ void copy_symlink(T&& existing_symlink, T&& new_symlink, error_code& ec);

+ template<NonLegacyPath T>
+ bool create_directories(T&& p);

+ template<NonLegacyPath T>
+ bool create_directories(T&& p, error_code& ec);

+ template<NonLegacyPath T>
+ bool create_directory(T&& p);

+ template<NonLegacyPath T>
+ bool create_directory(T&& p, error_code& ec);

+ template<NonLegacyPath T, NonLegacyPath U>
+ bool create_directory(T&& p, U&& attributes);

+ template<NonLegacyPath T, NonLegacyPath U>
+ bool create_directory(T&& p, U&& attributes, error_code& ec);

+ template<NonLegacyPath T, NonLegacyPath U>
+ void create_directory_symlink(T&& to, U&& new_symlink);

+ template<NonLegacyPath T, NonLegacyPath U>
+ void create_directory_symlink(T&& to, U&& new_symlink, error_code& ec);

+ template<NonLegacyPath T, NonLegacyPath U>
+ void create_hard_link(T&& to, U&& new_hard_link);

+ template<NonLegacyPath T, NonLegacyPath U>
+ void create_hard_link(T&& to, U&& new_hard_link, error_code& ec);

+ template<NonLegacyPath T, NonLegacyPath U>
+ void create_symlink(T&& to, U&& new_symlink);

+ template<NonLegacyPath T, NonLegacyPath U>
+ void create_symlink(T&& to, U&& new_symlink, error_code& ec);

+ template<NonLegacyPath T>
+ void current_path(T&& p);
template<NonLegacyPath T>
void current_path(T&& p, error_code& ec);

template<NonLegacyPath T, NonLegacyPath U>
bool equivalent(T&& p1, U&& p2);

template<NonLegacyPath T, NonLegacyPath U>
bool equivalent(T&& p1, U&& p2, error_code& ec);

template<NonLegacyPath T>
bool exists(T&& p);

template<NonLegacyPath T>
bool exists(T&& p, error_code& ec);

template<NonLegacyPath T>
uintmax_t file_size(T&& p);

template<NonLegacyPath T>
uintmax_t file_size(T&& p, error_code& ec);

template<NonLegacyPath T>
uintmax_t hard_link_count(T&& p);

template<NonLegacyPath T>
uintmax_t hard_link_count(T&& p, error_code& ec);

template<NonLegacyPath T>
bool is_block_file(T&& p);

template<NonLegacyPath T>
bool is_block_file(T&& p, error_code& ec);

template<NonLegacyPath T>
bool is_character_file(T&& p);

template<NonLegacyPath T>
bool is_character_file(T&& p, error_code& ec);

template<NonLegacyPath T>
bool is_directory(T&& p);

template<NonLegacyPath T>
bool is_directory(T&& p, error_code& ec);

template<NonLegacyPath T>
bool is_empty(T&& p);

template<NonLegacyPath T>
bool is_empty(T&& p, error_code& ec);

template<NonLegacyPath T>
bool is_fifo(T&& p);
template<NonLegacyPath T>
bool is_fifo(T&& p, error_code& ec);

template<NonLegacyPath T>
bool is_other(T&& p);

template<NonLegacyPath T>
bool is_other(T&& p, error_code& ec);

template<NonLegacyPath T>
bool is_regular_file(T&& p);

template<NonLegacyPath T>
bool is_regular_file(T&& p, error_code& ec);

template<NonLegacyPath T>
bool is_socket(T&& p);

template<NonLegacyPath T>
bool is_socket(T&& p, error_code& ec);

template<NonLegacyPath T>
bool is_symlink(T&& p);

template<NonLegacyPath T>
bool is_symlink(T&& p, error_code& ec);

template<NonLegacyPath T>
file_time_type last_write_time(T&& p);

template<NonLegacyPath T>
file_time_type last_write_time(T&& p, error_code& ec);

template<NonLegacyPath T>
void last_write_time(T&& p, file_time_type new_time);

template<NonLegacyPath T>
void last_write_time(T&& p, file_time_type new_time, error_code& ec);

template<NonLegacyPath T>
void permissions(T&& p, perms prms, perm_options opts=perm_options::replace);

template<NonLegacyPath T>
void permissions(T&& p, perms prms, error_code& ec);

template<NonLegacyPath T>
void permissions(T&& p, perms prms, perm_options opts, error_code& ec);

template<NonLegacyPath T>
path proximate(T&& p, error_code& ec);

template<NonLegacyPath T, NonLegacyPath U>
path proximate(T&& p, U&& base = current_path());
template<NonLegacyPath T, NonLegacyPath U>
path proximate(T&& p, U&& base, error_code& ec);

template<NonLegacyPath T>
path read_symlink(T&& p);

template<NonLegacyPath T>
path read_symlink(T&& p, error_code& ec);

template<NonLegacyPath T>
path relative(T&& p, error_code& ec);

template<NonLegacyPath T, NonLegacyPath U>
path relative(T&& p, U&& base = current_path());

template<NonLegacyPath T, NonLegacyPath U>
path relative(T&& p, U&& base, error_code& ec);

template<NonLegacyPath T>
bool remove(T&& p);

template<NonLegacyPath T>
bool remove(T&& p, error_code& ec);

template<NonLegacyPath T>
uintmax_t remove_all(T&& p);

template<NonLegacyPath T>
uintmax_t remove_all(T&& p, error_code& ec);

template<NonLegacyPath T, NonLegacyPath U>
void rename(T&& from, U&& to);

template<NonLegacyPath T, NonLegacyPath U>
void rename(T&& from, U&& to, error_code& ec);

template<NonLegacyPath T>
void resize_file(T&& p, uintmax_t size);

template<NonLegacyPath T>
void resize_file(T&& p, uintmax_t size, error_code& ec);

template<NonLegacyPath T>
space_info space(T&& p);

template<NonLegacyPath T>
space_info space(T&& p, error_code& ec);

template<NonLegacyPath T>
file_status status(T&& p);

template<NonLegacyPath T>
file_status status(T&& p, error_code& ec);
In 29.12.6 [fs.class.path] paragraph 6:

+ template<PathView T>
+ explicit path(T&& p);
+ template<PathView T>
+ path(T&& p, const locale& loc);
+ template<PathView T>
+ path& operator=(T&& p);
+ template<PathView T>
+ path& assign(T&& p);
+ template<PathView T>
+ path& operator/=(T&& p);
+ template<PathView T>
+ path& operator+=(T&& p);
+ format formatting()const noexcept;
+ template<PathView T>
+ path& replace_filename(T&& p);
+ template<PathView T>
+ path& replace_extension(T&& p);
+ template<PathView T>
+ friend path operator/ (const path& lhs, T&& rhs);
+ friend path operator/ (path&& lhs, path&& rhs);

[Note: The above rvalue ref path operator/ overload isn’t strictly needed for path view, but helps ameliorate the "my"/ "path"/ "literal" pattern you see in code which is currently extremely inefficient at runtime. – end note]
Note: The above rvalue ref path overload isn’t strictly needed for path view, but makes path more consistent with stringstream’s str(), and lets one write more efficient code. I’ll also be honest in saying that more than once in performance critical code I’ve written code such as \texttt{const\_cast\_string\_type&}(path.native()).\texttt{resize(N)} and then written a hexadecimal string directly into path’s underlying string representation. – end note

In 29.12.6.5.1 [fs.path.construct]:

+ \texttt{template}<\texttt{PathView T}> + \texttt{explicit\ path(T&& p);} + \texttt{Effects: Constructs an object of class path by an equivalent call to:}

```cpp
visit([&p] (auto sv) -> path {
  if constexpr(same_as<remove_cvref_t<decltype(sv)>, span<const byte>>)
  {
    // Implementation defined
  }
  else
  {
    return path(sv, p.formatting());
  }
}, p);
```

+ \texttt{template}<\texttt{PathView T}> + \texttt{path(T&& p, const locale& loc);} + \texttt{Effects: Constructs an object of class path by an equivalent call to:}

```cpp
visit([&p, &loc] (auto sv) -> path {
  if constexpr(same_as<remove_cvref_t<decltype(sv)>, span<const byte>>)
  {
    // Implementation defined
  }
  else
  {
    return path(sv, loc, p.formatting());
  }
```
+ format formatting() const noexcept;
+ Returns: The appropriate path separator format interpretation for the current path’s contents.

[Note: For brevity, I have not described the PathView 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. – end note]

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 CharT* b, size_type l, enum zero_termination zt, format fmt = path::binary_format) noexcept;
constexpr path_view_component(const byte* b, size_type l, enum zero_termination zt) noexcept;

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

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

template<class It, class End>
constexpr path_view_component(It b, End e, enum zero_termination zt, format fmt = path::binary_format) noexcept;
template<class It, class End>
constexpr path_view_component(It b, End e, enum zero_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_zero_termination() const noexcept;
constexpr enum zero_termination zero_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::zero_termination ZeroTermination,
    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 <enum path_view_component::zero_termination ZeroTermination,
class T = typename path::value_type,
class Allocator = default_rendered_path_allocator<T>,
size_type InternalBufferSize = default_internal_buffer_size>
constexpr rendered_path<ZeroTermination, T, Allocator, _internal_buffer_size>
render(path_view_component v, const locale& loc, Allocator allocate = Allocator());

template <enum path_view_component::zero_termination ZeroTermination,
class T = typename path::value_type,
class Allocator = default_rendered_path_allocator<T>,
size_type InternalBufferSize = default_internal_buffer_size>
constexpr rendered_path<ZeroTermination, T, Allocator, _internal_buffer_size>
render_zero_terminated(path_view_component v, Allocator allocate = Allocator());

private:
  union {
      const byte* bytestr_{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 zero_terminated_ : 1; // exposition only
uint16_t is_bytestr_ : 1; // exposition only
uint16_t is_wcharstr_ : 1; // exposition only
uint16_t is_charstr_ : 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;
template<class CharT>
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;

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 auto operator<=>(const CharT*, path_view_component) = delete;
template<class CharT>
inline constexpr auto operator<=>(basic_string_view<CharT>, path_view_component) = delete;
inline constexpr auto operator<=>(const byte*, path_view_component) = delete;
inline constexpr auto 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>& s, 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\(^2\).

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.
- generic_format causes only the generic path separator character (’/’) to delineate path com-
  ponents.
- 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 zero_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 imple-
mentation defined allocator.

Construction and assignment [fs.path_view_component.cons]

\(^2\)After much deliberation, LEWG chose 1,024 codepoints as a reasonable suggested default for most platforms.
```cpp
cconstexpr path_view_component() noexcept;

Effects: Constructs an object of class `path_view_component` which is empty.
Ensures: empty() == true and formatting() == format::unknown.
```

```cpp
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.
```

```cpp
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 zero_termination() == zero_terminated.
```

```cpp
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 zero_termination() == zero_terminated.
```

```cpp
template<class CharT>
constexpr path_view_component(const CharT* b, size_type l, enum zero_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 zero_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 zero_termination() == zt.
```

```cpp
constexpr path_view_component(const byte* b, size_type l, enum zero_termination zt) noexcept;
```
Expects: If \( \text{zt} \) is \text{zero_terminated}, then \([b, b + l]\) is a valid range and \(b[l] == \text{CharT}(0)\); otherwise \([b, b + l)\) is a valid range.

Effects: Constructs an object of class \text{path_view_component} which refers to a contiguous sequence of byte which begins at \(b\) and continues for \(l\) items.

Ensures: \text{formatting()} == \text{format::binary_format} and \text{zero_termination()} == \text{zt}.

\[
\begin{align*}
\text{template<class CharT>}
& \text{constexpr path_view_component(const CharT* b, format fmt = path::binary_format) noexcept;}
\end{align*}
\]

Constraints: \text{CharT} is any one of: \text{char}, \text{wchar_t}, \text{char8_t}, \text{char16_t}.

Expects: \([b, b + \text{char_traits<CharT>::length(b)}]\) is a valid range.

Effects: Equivalent to \text{path_view_component}(b, \text{char_traits<CharT>::length(b)}, \text{fmt}).

Ensures: \text{formatting()} == \text{fmt} and \text{zero_termination()} == \text{zero_terminated}.

Complexity: \(O(\text{char_traits<CharT>::length(b)})\).

\[
\begin{align*}
& \text{constexpr path_view_component(const byte* b) noexcept;}
\end{align*}
\]

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

Effects: Equivalent to \text{path_view_component}(b, (\text{size_type})(e - b)), if \text{memchr} were a \text{constexpr} available function.

Ensures: \text{formatting()} == \text{format::binary_format} and \text{zero_termination()} == \text{zero_terminated}.

Complexity: \(O(e - b)\).

[Note: 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. – end note]

\[
\begin{align*}
& \text{template<class CharT>}
& \text{constexpr path_view_component(basic_string_view<CharT> b, enum zero_termination zt,}
& \quad \text{format fmt = path::binary_format) noexcept;}
\end{align*}
\]

Constraints: \text{CharT} is any one of: \text{char}, \text{wchar_t}, \text{char8_t}, \text{char16_t}; if \text{zt} is \text{zero_terminated}, then \text{b.data()}[\text{b.size()}] == \text{CharT}(0).

Effects: Equivalent to \text{path_view_component}(\text{b.data()}, \text{b.size()}, \text{zt}, \text{fmt}).

Ensures: \text{formatting()} == \text{fmt} and \text{zero_termination()} == \text{zt}.

\[
\begin{align*}
& \text{constexpr path_view_component(span<const byte> b, enum zero_termination zt) noexcept;}
\end{align*}
\]
Constraints: If \( zt \) is zero_terminated, then \( b.data()[b.size()] == \text{byte}(0) \).

Effects: Equivalent to \( \text{path\_view\_component}(b.data(), b.size(), zt) \).

Ensures: \( \text{formatting()} == \text{format::binary\_format} \) and \( \text{zero\_termination()} == zt \).

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

**Constraints:**
1. It satisfies \text{contiguous\_iterator}.
2. End satisfies \text{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 zero_terminated, then \( *e == X(0) \). 

**Expects:**
1. If \( zt \) is zero_terminated, then \( [b, e] \) is a valid range, otherwise \( (b, e) \) is a valid range. 
2. It models \text{contiguous\_iterator}. 
3. End models \text{sized\_sentinel\_for<It>}. 

**Effects:** Equivalent to \( \text{path\_view\_component}(\text{to\_address(begin)}, \text{end\_begin}, zt, fmt) \).

**Ensures:** \( \text{formatting()} == \text{fmt} \) and \( \text{zero\_termination()} == zt \).

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

**Constraints:**
1. It satisfies \text{contiguous\_iterator}. 
2. End satisfies \text{sized\_sentinel\_for<It>}. 
3. iter\_value\_t<It> is byte. 
4. is\_convertible\_v<End, size\_type> is false. 
5. If \( zt \) is zero_terminated, then \( *e == \text{byte}(0) \). 

**Expects:**
1. If \( zt \) is zero_terminated, then \( [b, e] \) is a valid range, otherwise \( (b, e) \) is a valid range. 
2. It models \text{contiguous\_iterator}. 
3. End models \text{sized\_sentinel\_for<It>}. 

22
Effects: Equivalent to `path_view_component(to_address(begin), end - begin, zt)`.

Ensures: `formatting()== format::binary_format` and `zero_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_zero_termination() const noexcept;
```

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

```cpp
constexpr enum zero_termination zero_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())`.
Returns: True if extension() return a non-empty path view component.

Complexity: \(O(\text{native}_\text{size}())\).

```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 \([s + 1, \text{native}_\text{size}())\) unless \([s, \text{native}_\text{size}())\) 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: \(O(\text{native}_\text{size}())\).

[Note: The current normative wording for path::stem() is unclear how to handle "/foo/bar/..", so I chose for stem() to return ‘.’ and extension() to return ‘’ in this circumstance. – end note]

```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 \([s + 1, \text{native}_\text{size}())\) unless \([s, \text{native}_\text{size}())\) is ‘.’, otherwise then to one past the last element in the path view component; returns the portion of the path view component matching \([e, \text{native}_\text{size}())\).

Complexity: \(O(\text{native}_\text{size}())\).

```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: 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:

```cpp
path_view_component::rendered_path<T, Allocator, InternalBufferSize> zpath1(*this), zpath2(p);
p = zpath1.buffer, zpath1.length, this->formatting(), path2(zpath2.buffer, zpath2.length, p.formatting());
path1.compare(path2);
```
**Complexity:** \(O(\text{native}_\text{size}())\).

[Note: 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. – end note]

```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 locale& loc) 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:** Equivalent to `compare<T, Allocator, InternalBufferSize>` but where the implied `path_view_component::rendered_path` is constructed with an additional `loc` argument.

---

**Class `path_view_component::rendered_path`**

```cpp
namespace std::filesystem {
   template<enum path_view_component::zero_termination ZeroTermination,
      class T = typename path::value_type,
      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,
         const locale& loc,
```
Allocator allocate = Allocator();

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 const_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; /* 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 zero_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.

[Note: Objects of class path_view_component::rendered_path are typically expected to be instantiated upon the stack immediately preceding a system call, and destroyed shortly after return from a system call. The large default storage requirements of rendered_path therefore does not introduce the usual concerns, though we disable the copy constructor to prevent unintentional inefficiency. The move constructor remains available however, as in empirical usage it was found occasionally useful to pre-render an array of path views into an array of their syscall-ready forms e.g. if the path views' backing data would be about to disappear. Being able to store rendered_path inside STL containers is useful in this situation. – end note]

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 `ZeroTermination` is zero_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 = EOPNOTSUPP;
        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_zero_terminated();
    return ::open(zpath.c_str(), O_RDONLY);
}
```

[Note: Requiring users to always type out the zero termination they require may seem onerous, however defaulting the parameter to zero terminate would lead to accidental memory copies purely and solely just to zero terminate, which eliminates much of the advantage of using path views. It is correct that the primary filesystem APIs on the two major platforms both require zero termination, however there are many other uses of paths e.g. with ZIP archive libraries, Java JNI, etc with APIs that take a lengthed path. Many of the secondary filesystem APIs on the two major platforms also use lengthed not zero terminated paths e.g. Microsoft Windows NT kernel API, which if using then memory copies can be completely avoided. – end note]

**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, 
    const locale& loc, 
    Allocator allocate = Allocator());
```
Effects:

- If `value_type` is byte, `size()` will return `v.native_size()`. If `zero_termination` is `zero_terminated` and `v.zero_termination()` is `not_zero_terminated`:

  - 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 `zero_termination` is `zero_terminated`, and either `(v.native_size() + v.zero_terminated()) != (size() + 1) * sizeof(value_type)` is true or `v.zero_termination()` is `not_zero_terminated`:

  - 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.

  [Note: The `(v.native_size() + v.has_zero_termination()) != (size() + 1) * sizeof(value_type)` is to enable pass-through 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`. – end note]

- 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 zero_termination is zero_terminated and v.zero_termination() is not_zero_terminated, 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 using loc, replacing any generic path separators with native path separators if v.formatting() allows that, zero value_type terminating the reencoded buffer if zero_termination is zero_terminated. data() shall return that reencoded path, and size() shall be the number of elements output, excluding any zero termination appended. It is defined by loc what occurs if the backing data contains invalid codepoints for its declared encoding.

[Note: It should be noted that it is not always possible to directly use loc for a reencode between certain combinations of source and destination encoding on some platforms. For example, on Microsoft Windows, because the narrow system encoding is runtime configurable and therefore a system API is always invoked to perform a reencode to the narrow system encoding, one cannot use loc directly. Rather, one uses loc to reencode from the source encoding into wchar_t, then invoke the Microsoft Windows system API to reencode from wchar_t to the narrow system encoding i.e. there are two reencode passes here. Reading the current normative wording for how path uses loc, this appears to be a valid interpretation of how path already works, though I do note path’s wording in this area is somewhat ambiguous currently.

Another thing to note is that if the user does not use the loc consuming overload, we need to allow implementations to use any as-if equivalent reencoding mechanism to std::locale(). This is because some platforms implement system APIs for reencoding strings which have far better performance than standard library facilities, and we ought to not prevent implementations making use of those. The current path normative wording also appears to allow this optimisation. – end note]
**Effects:** The member functions `data()` and `size()` return values as for the preceding overload, using implementation defined mechanisms for performing reencodings if that is necessary. Implementations are required to throw an exception if the backing data is `char8_t` or `char16_t` and the backing data contains an invalid codepoint for UTF-8 or UTF-16 respectively.

**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]

```cpp
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.

```cpp
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.

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

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

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

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

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

**Constraints:** `T` is any one of: `char`, `wchar_t`, `char8_t`, `char16_t`; ZeroTermination is `zero_termination::zero_terminated`.

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

**Non-member comparison functions [fs.path_view_component.comparison]**
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 \texttt{memcmp} is used to compare the backing bytes of both path view components for equality and ordering.

\[\text{Effect:} \quad \text{If the native sizes are unequal, the path view components are considered unequal. If the backing bytes are of different encoding, the path view components are considered unequal. Otherwise a comparison equivalent to } \texttt{memcmp} \text{ is used to compare the backing bytes of both path view components for equality and ordering.}\]

\textit{Note:} This is intentionally a ‘shallow’ equality comparison intended for use in maps etc, it doesn’t do expensive \texttt{compare()}.

[\textit{end note}]
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.

[Note: Experience of how people use path views in the real world found that much unintended performance loss derives from people comparing path views to string literals, because of the potential extremely expensive (relatively speaking) reencode and dynamic memory allocation per comparison due to the implicit conversion. Even this paper's author, who you would think would really know better, got stung by this on more than one occasion. So the decision was taken to prevent such constructs compiling at all, which is very unfortunate, but it does force the user to write much better code e.g. `visit()` with an overload set for `operator()(basic_string_view<CharT>)` for each possible backing data encoding. – end note]

Non-member functions [fs.path_view_component.comparison]

```cpp
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)`.

```cpp
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.

```cpp
template<class charT, class traits>
basic_ostream<charT, traits>& operator<<(basic_ostream<charT, traits>& s, path_view_component v);
```

Effects: Equivalent to:

```cpp
template<class charT, class traits>
basic_ostream<charT, traits>& operator<<(basic_ostream<charT, traits>& s, path_view_component v)
{
```

33
namespace std::filesystem {
    class path_view : public path_view_component {
public:
    using const_iterator = /* implementation defined */;
    using iterator = /* implementation defined */;
    using input_type = remove_cvref_t<decltype(sv)>>;
    using output_type = basic_ostream<charT, traits>;
    if constexpr(same_as<input_type, span<const byte>>)
    {
    /* Implementation defined. Microsoft Windows requires the following
    textualisation for FILE_ID_DESCRIPTOR.ObjectId keys which are guids:
    
    "\{7ecf65a0-4b78-5f9b-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.
    */
    }
    else
    {
    // Possibly reencode to ostream’s character type
    path_view_component::rendered_path<typename output_type::char_type> zbuff(v, path_view_component
    ::not_zero_terminated);
    return quoted(s.write(zbuff.buffer, zbuff.length));
    }, v);
    return visit([&s, &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>>)
        {
            /* Implementation defined. Microsoft Windows requires the following
            textualisation for FILE_ID_DESCRIPTOR.ObjectId keys which are guids:
            
            "\{7ecf65a0-4b78-5f9b-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.
            */
        }
        else
        {
            // Possibly reencode to ostream’s character type
            path_view_component::rendered_path<typename output_type::char_type> zbuff(v, path_view_component
            ::not_zero_terminated);
            return quoted(s.write(zbuff.buffer, zbuff.length));
        }
    }, v);
}

Returns: s.

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.
using reverse_iterator = /* implementation defined */;
using const_reverse_iterator = /* implementation defined */;
using difference_type = /* implementation defined */;

public:
   // Constructors and destructors
   constexpr path_view() noexcept;
   path_view(path_view_component p, format fmt = path::auto_format) noexcept;
   path_view(const path& p) noexcept;
   template<class CharT>
   constexpr path_view(const basic_string<CharT>,
                       format fmt = path::auto_format) noexcept;
   template<class CharT>
   constexpr path_view(const CharT* b, size_type l, enum zero_termination zt,
                       format fmt = path::auto_format) noexcept;
   constexpr path_view(const byte* b, size_type l, enum zero_termination zt) noexcept;
   template<class CharT>
   constexpr path_view(const CharT* b, format fmt = path::auto_format) noexcept;
   constexpr path_view(const byte* b) noexcept;
   template<class CharT>
   constexpr path_view(basic_string_view<CharT> b, enum zero_termination zt,
                       format fmt = path::auto_format) noexcept;
   constexpr path_view(span<const byte> b, enum zero_termination zt) noexcept;
   template<class It, class End>
   constexpr path_view(It b, End e, enum zero_termination zt,
                       format fmt = path::auto_format) noexcept;
   template<class It, class End>
   constexpr path_view(It b, End e, enum zero_termination zt) noexcept;
   constexpr path_view(const path_view&) = default;
   constexpr path_view(path_view&&) = default;
   constexpr ~path_view() = default;

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

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

   // Query
   constexpr bool has_root_name() const noexcept;
   constexpr bool has_root_directory() const noexcept;
   constexpr bool has_root_path() const noexcept;
   constexpr bool has_relative_path() const noexcept;
   constexpr bool has_parent_path() const noexcept;
   constexpr bool has_filename() const noexcept;
   constexpr bool is_absolute() const noexcept;
   constexpr bool is_relative() const noexcept;
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]

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```cpp
constexpr bool is_absolute() const noexcept;
```
Returns: True if the path view contains an absolute path after interpretation by formatting().

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

Returns: True if is_absolute() is false.

```cpp
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()).*

```cpp
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()).*

```cpp
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()).*

```cpp
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()).*

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

Complexity: O(native_size()).

```cpp
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()).

```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 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;
```

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()).

In 29.12.8.1 [fs.enum.path.format] paragraph 1:

+ binary_format The binary pathname format.

4 Acknowledgements

Thanks to Billy O’Neal for making me rewrite most of my first attempt at this normative wording. Thanks to Jonathan Wakely for provided detailed notes as only the chair of LWG would perceive. Thanks to Corentin and Peter Dimov for additional feedback.

R5 of this paper would not have happened in a timely fashion without my current client MayStreet London Stock Exchange Group permitting me a few work hours to complete the R5 revision. My thanks to them for enabling this paper revision, it has been some years since the last one!
5 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