D1030R4 draft 1: `std::filesystem::path_view`

Document #:	D1030R4 draft 1
Date:	2020-10-01
Project:	Programming Language C++ Library Evolution Working Group

Reply-to:	Niall Douglas <s_sourceforge@nedprod.com>

A proposed wording for a `std::filesystem::path_view`, a non-owning view of explicitly unencoded or encoded character sequences in the format of a local 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.

As this proposal now targets the IS, from R4 onwards it consists of just normative wording. If you wish to understand the design rationale and evolution of the design, please consult R3 or earlier (`https://wg21.link/P1030R3`).

If you wish to use an implementation right now, a mostly-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.

Changes since R3 due to LEWG feedback:

- `span<byte>` for binary keys.
- `compare<>` needs to work like path's compare e.g. collapse multiple separators.

Contents

1 Known backward source incompatibilities

1.1 Functions currently accepting `filesystem::path` would no longer implicitly accept `char32_t` inputs

Under this revision of this proposal, all functions currently consuming `filesystem::path` now consume a `filesystem::path_view` instead. This can significantly improve performance if source encoding matches filesystem encoding, usually avoids a dynamic memory allocation and free otherwise, and it also enables existing `path` consuming standard C++ facilities to accept binary keys identifying files on supporting filesystems and storage devices.

Path views do not construct from `char32_t` as such a source for a path always requires a copy and encoding translation (this design choice was voted upon by LEWG, and agreed with). That means that under the present wording, C++ 20 standards conforming code such as the following would no longer compile:

```
// libstdc++ does not compile this
// MSVC and libc++ does compile this
std::ofstream s(U"meow.txt");  // a char32_t string literal
```

Complete backwards source compatibility could be retained by adding a `char32_t` accepting overload everywhere there is a `path_view` consuming function. However it is believed that the amount of C++ code in the ecosystem which supplies `char32_t` strings and relies on **implicit** `path` construction from `char32_t` approaches zero:

1. A search of github for source files containing 'path' and 'char32_t' found zero results.
2. A search of `https://codesearch.isocpp.org/` found only standard library unit test code ensuring paths can construct from `char32_t`.
3. A search of `https://grep.app/` for 'path char32_t' also found zero results.

I want to *stress* that `path` itself remains able to implicitly construct from `char32_t`. So after this paper in its current form were applied, the code above would need to be changed to this to compile:

```
// Already needed if using libstdc++ which won't implicitly construct
// paths from char32_t input
std::ofstream s(std::filesystem::path(U"meow.txt"));
```

It should be noted that this has `path` convert `U"meow.txt"` into the native filesystem encoding, from which `path_view` borrows.

2 Delta from N4861

The following normative wording delta is against `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/n4861.pdf`.

In 17.3.2 [version.syn] paragraph 2:

```
#define __cpp_lib_filesystem 201703L 202010L //also in <filesystem>
```

In 29.9.1 [filebuf] paragraph 3:

~~In subclause 29.9, member functions taking arguments of `const filesystem::path::value_type*` are only be provided on systems where `filesystem::path::value_type` (29.11.7) is not char.~~

In 29.9.2 [filebuf]:

```
- basic_filebuf* open(const char* s, ios_base::openmode mode);
- basic_filebuf* open(const filesystem::path::value_type* s, ios_base::openmode mode);
- basic_filebuf* open(const string& s, ios_base::openmode mode);
- basic_filebuf* open(const filesystem::path& s, ios_base::openmode mode);
+ basic_filebuf* open(filesystem::path_view s, ios_base::openmode mode);
```

> [*Note:* Implementations would still offer linkage for the removed overloads for backwards binary compatibility, but at source code level the only way of supplying a path becomes a path view. As path views construct from all of the above, source code compatibility is retained. – end note]

In 29.9.2.3 [filebuf.members] paragraph 2:

```
- basic_filebuf* open(const char* s, ios_base::openmode mode);
- basic_filebuf* open(const filesystem::path::value_type* s, ios_base::openmode mode);
+ basic_filebuf* open(filesystem::path_view s, ios_base::openmode mode);
```

\+ *Expects:* s points to a non-empty path view.

In 29.9.3 [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);
- explicit basic_ifstream(const string& s, ios_base::openmode mode = ios_base::in);
- explicit basic_ifstream(const filesystem::path& s, ios_base::openmode mode = ios_base::in);
+ explicit basic_ifstream(filesystem::path_view s, ios_base::openmode mode = ios_base::in);
```

```
- void open(const char* s, ios_base::openmode mode = ios_base::in);
- void open(const filesystem::path::value_type* s, ios_base::openmode mode = ios_base::in);
- void open(const string& s, ios_base::openmode mode = ios_base::in);
- void open(const filesystem::path& s, ios_base::openmode mode = ios_base::in);
+ void open(filesystem::path_view s, ios_base::openmode mode = ios_base::in);
```

[*Note:* Implementations would still offer linkage for the removed overloads for backwards binary compatibility, but at source code level the only way of supplying a path becomes a path view. As path views construct from all of the above, source code compatibility is retained. – end note]

In 29.9.3.1 [ifstream.cons] paragraph 2:

```
- explicit basic_ifstream(const char* s, ios_base::openmode mode = ios_base::in);
- explicit basic_ifstream(const filesystem::path::value_type* s, ios_base::openmode mode = ios_base::in);
- explicit basic_ifstream(const string& s, ios_base::openmode mode = ios_base::in);
- explicit basic_ifstream(const filesystem::path& s, ios_base::openmode mode = ios_base::in);
+ explicit basic_ifstream(filesystem::path_view s, ios_base::openmode mode = ios_base::in);
```

In 29.9.3.3 [ifstream.members] paragraph 3:

```
- void open(const char* s, ios_base::openmode mode = ios_base::in);
- void open(const filesystem::path::value_type* s, ios_base::openmode mode = ios_base::in);
- void open(const string& s, ios_base::openmode mode = ios_base::in);
- void open(const filesystem::path& s, ios_base::openmode mode = ios_base::in);
+ void open(filesystem::path_view s, ios_base::openmode mode = ios_base::in);
```

In 29.9.4 [ofstream]:

```
- explicit basic_ofstream(const char* s, ios_base::openmode mode = ios_base::out);
- explicit basic_ofstream(const filesystem::path::value_type* s, ios_base::openmode mode = ios_base::out);
- explicit basic_ofstream(const string& s, ios_base::openmode mode = ios_base::out);
- explicit basic_ofstream(const filesystem::path& s, ios_base::openmode mode = ios_base::out);
+ explicit basic_ofstream(filesystem::path_view s, ios_base::openmode mode = ios_base::out)
;
```

```
- 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);
- 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 open(filesystem::path_view s, ios_base::openmode mode = ios_base::out);
```

[*Note:* Implementations would still offer linkage for the removed overloads for backwards binary compatibility, but at source code level the only way of supplying a path becomes a path view. As path views construct from all of the above, source code compatibility is retained. – end note]

In 29.9.4.1 [ofstream.cons] paragraph 2:

```
- explicit basic_ofstream(const char* s, ios_base::openmode mode = ios_base::out);
- explicit basic_ofstream(const filesystem::path::value_type* s, - ios_base::openmode mode = ios_base::out);
- explicit basic_ofstream(const string& s, ios_base::openmode mode = ios_base::out);
- explicit basic_ofstream(const filesystem::path& s, - ios_base::openmode mode = ios_base::out);
+ explicit basic_ofstream(filesystem::path_view s, ios_base::openmode mode = ios_base::out)
;
```

In 29.9.4.3 [ofstream.members] paragraph 3:

```
- void open(const char* s, ios_base::openmode mode = ios_base::out);
- void open(const filesystem::path::value_type* s, ios_base::openmode mode = ios_base::out);
- 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 open(filesystem::path_view s, ios_base::openmode mode = ios_base::out);
```

In 29.9.5 [fstream]:

```
- explicit basic_fstream(const char* s, - ios_base::openmode mode = ios_base::in | ios_base::out);
- explicit basic_fstream(const filesystem::path::value_type* s, - ios_base::openmode mode = ios_base::in|ios
- explicit basic_fstream(const string& s, - ios_base::openmode mode = ios_base::in | ios_base::out);
- explicit basic_fstream(const filesystem::path& s, - ios_base::openmode mode = ios_base::in | ios_base::out
```

```
+ explicit basic_fstream(filesystem::path_view s, ios_base::openmode mode = ios_base::in |
ios_base::out);

- 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)
- 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(filesystem::path_view s, ios_base::openmode mode = ios_base::in | ios_base::out
);
```

[*Note:* Implementations would still offer linkage for the removed overloads for backwards binary compatibility, but at source code level the only way of supplying a path becomes a path view. As path views construct from all of the above, source code compatibility is retained. – end note]

In 29.9.5.1 [fstream.cons] paragraph 2:

```
- explicit basic_fstream(const char* s, - ios_base::openmode mode = ios_base::in | ios_base::out);
- explicit basic_fstream(const filesystem::path::value_type* s, - ios_base::openmode mode = ios_base::in|ios
- explicit basic_fstream(const string& s, - ios_base::openmode mode = ios_base::in | ios_base::out);
- explicit basic_fstream(const filesystem::path& s, - ios_base::openmode mode = ios_base::in | ios_base::out
+ explicit basic_fstream(filesystem::path_view s, ios_base::openmode mode = ios_base::in |
ios_base::out);
```

In 29.9.5.3 [fstream.members] paragraph 2:

```
- 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)
- 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(filesystem::path_view s, ios_base::openmode mode = ios_base::in | ios_base::out
);
```

In 29.11.5 [fs.filesystem.syn]:

```
+ class path_view;
path absolute(const path& p path_view p);
```

```
path absolute(const path& p path_view p, error_code& ec);

path canonical(const path& p path_view p);

path canonical(const path& p path_view p, error_code& ec);

void copy(const path& from, const path& to path_view from, path_view to);

void copy(const path& from, const path& to path_view from, path_view to, error_code& ec);

void copy(const path& from, const path& to path_view from, path_view to, copy_options options
);

void copy(const path& from, const path& to path_view from, path_view to, copy_options options
, error_code& ec);

bool copy_file(const path& from, const path& to path_view from, path_view to);

bool copy_file(const path& from, const path& to path_view from, path_view to, error_code&
ec);

bool copy_file(const path& from, const path& to path_view from, path_view to, copy_options
options);

bool copy_file(const path& from, const path& to path_view from, path_view to, copy_options
options, error_code& ec);

void copy_symlink(const path& existing_symlink, const path& new_symlink path_view exist-
ing_symlink, path_view new_symlink);

void copy_symlink(const path& existing_symlink, const path& new_symlink path_view exist-
ing_symlink, path_view new_symlink, error_code& ec)noexcept;

bool create_directories(const path& p path_view p);

bool create_directories(const path& p path_view p, error_code& ec);

bool create_directory(const path& p path_view p);

bool create_directory(const path& p path_view p, error_code& ec)noexcept;

bool create_directory(const path& p, const path& attributes path_view p, path_view attributes)
;

bool create_directory(const path& p, const path& attributes path_view p, path_view attributes,
 error_code& ec)noexcept;

void create_directory_symlink(const path& to, const path& new_symlink path_view to, path_view
new_symlink);

void create_directory_symlink(const path& to, const path& new_symlink path_view to, path_view
new_symlink, error_code& ec)noexcept;

void create_hard_link(const path& to, const path& new_hard_link path_view to, path_view
new_hard_link);
```

```
void create_hard_link(const path& to, const path& new_hard_link path_view to, path_view
new_hard_link, error_code& ec)noexcept;

void create_symlink(const path& to, const path& new_symlink path_view to, path_view new_symlink)
;

void create_symlink(const path& to, const path& new_symlink path_view to, path_view new_symlink,
 error_code& ec)noexcept;

path current_path();

path current_path(error_code& ec);

void current_path(const path& p path_view p);

void current_path(const path& p path_view p, error_code& ec)noexcept;

bool equivalent(const path& p1, const path& p2 path_view p1, path_view p2);

bool equivalent(const path& p1, const path& p2 path_view p1, path_view p2, error_code& ec)
noexcept;

bool exists(file_status s)noexcept;

bool exists(const path& p path_view p);

bool exists(const path& p path_view p, error_code& ec)noexcept;

uintmax_t file_size(const path& p path_view p);

uintmax_t file_size(const path& p path_view p, error_code& ec)noexcept;

uintmax_t hard_link_count(const path& p path_view p);

uintmax_t hard_link_count(const path& p path_view p, error_code& ec)noexcept;

bool is_block_file(file_status s)noexcept;

bool is_block_file(const path& p path_view p);

bool is_block_file(const path& p path_view p, error_code& ec)noexcept;

bool is_character_file(file_status s)noexcept;

bool is_character_file(const path& p path_view p);

bool is_character_file(const path& p path_view p, error_code& ec)noexcept;

bool is_directory(file_status s)noexcept;

bool is_directory(const path& p path_view p);

bool is_directory(const path& p path_view p, error_code& ec)noexcept;

bool is_empty(const path& p path_view p);

bool is_empty(const path& p path_view p, error_code& ec);

bool is_fifo(file_status s)noexcept;
```

```
bool is_fifo(~~const path& p~~ path_view p);
bool is_fifo(~~const path& p~~ path_view p, error_code& ec)noexcept;
bool is_other(file_status s)noexcept;
bool is_other(~~const path& p~~ path_view p);
bool is_other(~~const path& p~~ path_view p, error_code& ec)noexcept;
bool is_regular_file(file_status s)noexcept;
bool is_regular_file(~~const path& p~~ path_view p);
bool is_regular_file(~~const path& p~~ path_view p, error_code& ec)noexcept;
bool is_socket(file_status s)noexcept;
bool is_socket(~~const path& p~~ path_view p);
bool is_socket(~~const path& p~~ path_view p, error_code& ec)noexcept;
bool is_symlink(file_status s)noexcept;
bool is_symlink(~~const path& p~~ path_view p);
bool is_symlink(~~const path& p~~ path_view p, error_code& ec)noexcept;
file_time_type last_write_time(~~const path& p~~ path_view p);
file_time_type last_write_time(~~const path& p~~ path_view p, error_code& ec)noexcept;
void last_write_time(~~const path& p~~ path_view p, file_time_type new_time);
void last_write_time(~~const path& p~~ path_view p, file_time_type new_time, error_code& ec)noexcept
;
void permissions(~~const path& p~~ path_view p, perms prms, perm_options opts=perm_options::replace
);
void permissions(~~const path& p~~ path_view p, perms prms, error_code& ec)noexcept;
void permissions(~~const path& p~~ path_view p, perms prms, perm_options opts, error_code& ec);
path proximate(~~const path& p~~ path_view p, error_code& ec);
path proximate(~~const path& p, const path& base~~ path_view p, path_view base= current_path());
path proximate(~~const path& p, const path& base~~ path_view p, path_view base, error_code& ec);
path read_symlink(~~const path& p~~ path_view p);
path read_symlink(~~const path& p~~ path_view p, error_code& ec);
path relative(~~const path& p~~ path_view p, error_code& ec);
path relative(~~const path& p, const path& base~~ path_view p, path_view base= current_path());
path relative(~~const path& p, const path& base~~ path_view p, path_view base, error_code& ec);
```

```
bool remove(const path& p path_view p);

bool remove(const path& p path_view p, error_code& ec)noexcept;

uintmax_t remove_all(const path& p path_view p);

uintmax_t remove_all(const path& p path_view p, error_code& ec)noexcept;

void rename(const path& from, const path& to path_view from, path_view to);

void rename(const path& from, const path& to path_view from, path_view to, error_code& ec)
noexcept;

void resize_file(const path& p path_view p, uintmax_t size);

void resize_file(const path& p path_view p, uintmax_t size, error_code& ec)noexcept;

space_info space(const path& p path_view p);

space_info space(const path& p path_view p, error_code& ec)noexcept;

file_status status(const path& p path_view p);

file_status status(const path& p path_view p, error_code& ec)noexcept;

bool status_known(file_status s)noexcept;

file_status symlink_status(const path& p path_view p);

file_status symlink_status(const path& p path_view p, error_code& ec)noexcept;

path temp_directory_path();

path temp_directory_path(error_code& ec);

path weakly_canonical(const path& p path_view p);

path weakly_canonical(const path& p path_view p, error_code& ec);
```

> [*Note:* Implementations would still offer linkage for the removed overloads for backwards binary compatibility, but at source code level the only way of supplying a path becomes a path view. As path views construct from all of the above, source code compatibility is retained. – end note]

In 29.11.7 [fs.class.path] paragraph 7:

```
+ explicit path(path_view p);

+ path(path_view p, const std::locale& loc);

+ path& operator=(path_view p);

+ path& assign(path_view p);

path& operator/=(const path& p path_view p);

path& operator+=(const path& p path_view p);
```

- path& operator+=(const string_type& x);

- path& operator+=(const basic_string_view<value_type> x);

- path& operator+=(const value_type* x);

path& replace_filename(const path& replacement path_view replacement);

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

friend path operator/ (const path& lhs, const path& rhs path_view rhs);

+ friend path operator/ (path&& lhs, path_view rhs);

+ friend path operator/ (path_view lhs, path_view rhs);

int compare(const path& p path_view p) const noexcept;

- int compare(const string_type& s)const

- int compare(const basic_string_view<value_type>& s)const

- int compare(const value_type* s)const

path lexically_relative(const path& base path_view base) const;

path lexically_proximate(const path& base path_view base) const;

> [*Note:* Implementations would still offer linkage for the removed overloads for backwards binary compatibility. Path views construct from all of the removed overloads, so source code compatibility is retained. The rvalue ref `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]

In 29.11.7.4.1 [fs.path.construct]:

+ explicit path(path_view p);

+ *Effects:* Constructs an object of class `path` by an as-if call of:

```
path_view::c_str<path::value_type> temporary(p);
path(basic_string_view<path::value_type>(temporary.buffer, temporary.length));.
```

+ path(path_view p, const std::locale& loc);

+ *Effects:* Constructs an object of class `path` by an as-if call of:

```
path_view::c_str<path::value_type> temporary(p, loc);
path(basic_string_view<path::value_type>(temporary.buffer, temporary.length), loc);.
```

> [*Note:* For brevity, I have not repeated the path to path view conversion changes already described. The changes are very mechanistic: for all inputs which are a const lvalue

reference or const character pointer e.g. `const path&`, replace with `path_view`. – 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 required to be trivially copyable.

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

```
namespace std::filesystem {
  class path_view_component {
  public:
    using size_type = path::string_type::size_type;
    static constexpr path::value_type preferred_separator = path::preferred_separator;
    static constexpr size_t default_internal_buffer_size = /* implementation defined */;

    using format = path::format;

    enum zero_termination {
      zero_terminated,
      not_zero_terminated
    };

    template<class T>
    /* implementation defined */ default_deleter = /* implementation defined */;

    // Constructors and destructor
    constexpr path_view_component() = default;

    path_view_component(const path &p) noexcept;
    template<class CharT>
    constexpr path_view_component(const basic_string<CharT>& s,
                                  format fmt = path::auto_format) noexcept;

    template<class CharT>
    constexpr path_view_component(const CharT* b, size_type l, zero_termination zt,
                                  format fmt = path::auto_format) noexcept;
    constexpr path_view_component(const byte* b, size_type l, zero_termination zt) noexcept;

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

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

```
39
40      template<class It, class End>
41      constexpr path_view_component(It b, End e, zero_termination zt,
42                                    format fmt = path::auto_format) noexcept;
43      template<class It, class End>
44      constexpr path_view_component(It b, End e, zero_termination zt) noexcept;
45
46      constexpr path_view_component(const path_view_component&) = default;
47      constexpr path_view_component(path_view_component&&) = default;
48      constexpr ~path_view_component() = default;
49
50      // Assignments
51      constexpr path_view_component &operator=(const path_view_component&) = default;
52      constexpr path_view_component &operator=(path_view_component&&) = default;
53
54      // Modifiers
55      constexpr void swap(path_view_component& o) noexcept;
56
57      // Query
58      [[nodiscard]] constexpr bool empty() const noexcept;
59      constexpr size_type native_size() const noexcept;
60      constexpr format formatting() const noexcept;
61      constexpr bool zero_terminated() const noexcept;
62      constexpr bool has_stem() const noexcept;
63      constexpr bool has_extension() const noexcept;
64
65      constexpr path_view_component stem() const noexcept;
66      constexpr path_view_component extension() const noexcept;
67
68      // Comparison
69      template<class T = typename path::value_type,
70               class Deleter = default_deleter<T[]>,
71               size_type InternalBufferSize = default_internal_buffer_size>
72      constexpr int compare(path_view_component p) const;
73      template<class T = typename path::value_type,
74               class Deleter = default_deleter<T[]>,
75               size_type InternalBufferSize = default_internal_buffer_size>
76      constexpr int compare(const path& p) const;
77      template<class T = typename path::value_type,
78               class Deleter = default_deleter<T[]>,
79               size_type InternalBufferSize = default_internal_buffer_size,
80               class CharT>
81      constexpr int compare(const basic_string<CharT>& s, format fmt = path::auto_format) const;
82      template<class T = typename path::value_type,
83               class Deleter = default_deleter<T[]>,
84               size_type InternalBufferSize = default_internal_buffer_size,
85               class CharT>
86      constexpr int compare(const CharT* s, format fmt = path::auto_format) const;
87      template<class T = typename path::value_type,
88               class Deleter = default_deleter<T[]>,
89               size_type InternalBufferSize = default_internal_buffer_size,
90               class CharT>
91      constexpr int compare(basic_string_view<CharT> s, format fmt = path::auto_format) const;
92      template<class T = typename path::value_type,
93               class Deleter = default_deleter<T[]>,
94               size_type InternalBufferSize = default_internal_buffer_size>
```

```
    constexpr int compare(span<const byte> s) const;

    template<class T = typename path::value_type,
             class Deleter = default_deleter<T[]>,
             size_type InternalBufferSize = default_internal_buffer_size>
    constexpr int compare(const path& p, const locale& loc) const;
    template<class T = typename path::value_type,
             class Deleter = default_deleter<T[]>,
             size_type InternalBufferSize = default_internal_buffer_size,
             class CharT>
    constexpr int compare(const basic_string<CharT>& s, const locale& loc,
                          format fmt = path::auto_format) const;
    template<class T = typename path::value_type,
             class Deleter = default_deleter<T[]>,
             size_type InternalBufferSize = default_internal_buffer_size>
    constexpr int compare(path_view_component p, const locale &loc) const;
    template<class T = typename path::value_type,
             class Deleter = default_deleter<T[]>,
             size_type InternalBufferSize = default_internal_buffer_size,
             class CharT>
    constexpr int compare(const CharT* s, const locale& loc,
                          format fmt = path::auto_format) const;
    template<class T = typename path::value_type,
             class Deleter = default_deleter<T[]>,
             size_type InternalBufferSize = default_internal_buffer_size,
             class CharT>
    constexpr int compare(basic_string_view<CharT> s, const locale& loc,
                          format fmt = path::auto_format) const;
    template<class T = typename path::value_type,
             class Deleter = default_deleter<T[]>,
             size_type InternalBufferSize = default_internal_buffer_size>
    constexpr int compare(span<const byte> s) const;

    // Conversion
    template<class T = typename path::value_type,
             class Deleter = default_deleter<T[]>,
             size_type InternalBufferSize = default_internal_buffer_size>
    struct c_str;

  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
    unsigned _zero_terminated : 1;    // exposition only
    unsigned _is_bytestr : 1;         // exposition only
    unsigned _is_charstr : 1;         // exposition only
    unsigned _is_wcharstr : 1;        // exposition only
    unsigned _is_char8str : 1;        // exposition only
    unsigned _is_char16str : 1;       // exposition only
    unsigned _format : 2;             // exposition only
```

```
  };

  // 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;

  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==(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!=(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!=(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>& 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::c_str` instance, which is usually instantiated onto the stack. It ought to be defined to a little more than the typical lengths of filesystem path on that platform[1].

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

[1] After much deliberation, LEWG chose 1,024 characters as a reasonable suggested default for most platforms.

- `native_format` causes only the native path separator character to delineate path components.
- `generic_format` causes only the generic path separator character ('/') to delineate path components.
- `binary_format` causes no delineation of path components at all in the backing data.
- `auto_format` causes *both* the native and generic path separators to delineate path components (and backing data may contain a mix of both).

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

`default_deleter<T>` is an implementation defined dynamic allocation deleter meeting the specification of `default_delete<T>`.

Construction and assignment [fs.path_view_component.cons]

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

Ensures: `empty()== true`.

```
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.

```
template<class CharT>
constexpr path_view_component(const basic_string<CharT>& s,
                              format fmt = path::auto_format) noexcept;
```

Constraints: `is_same_v<CharT, X>` is true for `X` being any one of: `char`, `wchar_t`, `char8_t`, `char16_t`.

Effects: Constructs an object of class `path_view_component` which refers to a zero terminated contiguous sequence of one of `char`, `wchar_t`, `char8_t` or `char16_t` which begins at `s.c_str()` and continues for `s.size()` items.

Ensures: `formatting()== fmt`.

```
template<class CharT>
constexpr path_view_component(const CharT* b, size_type l, zero_termination zt,
                              format fmt = path::auto_format) noexcept;
```

Constraints: `is_same_v<CharT, X>` is true for `X` being 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`.

```
    constexpr path_view_component(const byte* b, size_type l, zero_termination zt) noexcept;
```

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 `byte` which begins at `b` and continues for `l` items.

Ensures: `formatting()== format::binary_format`.

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

Constraints: `is_same_v<CharT, X>` is true for `X` being any one of: `char`, `wchar_t`, `char8_t`, `char16_t`.

Expects: `[b, b + char_traits<CharT>::length(b)]` is a valid range.

Effects: As if `path_view_component(b, char_traits<CharT>::length(b), fmt)`.

Ensures: `formatting()== fmt`.

Complexity: `O(char_traits<CharT>::length(b))`.

```
    constexpr path_view_component(const byte* b) noexcept;
```

Expects: `[b, memchr(b, 0)]` is a valid range.

Effects: As if `path_view_component(b, (size_type)(memchr(b, 0)- b))`, if `memchr` were a `constexpr` available function.

Ensures: `formatting()== format::binary_format`.

Complexity: `O(memchr(b, 0)- 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, implementation defined effects will occur. – end note]

```
    template<class CharT>
    constexpr path_view_component(basic_string_view<CharT> b, zero_termination zt,
                                  format fmt = path::auto_format) noexcept;
```

Constraints: `is_same_v<CharT, X>` is true for `X` being any one of: `char`, `wchar_t`, `char8_t`, `char16_t`; if `zt` is `zero_terminated`, then `b.data()[b.size()] == CharT(0)`.

Effects: As if `path_view_component(b.data(), b.size(), zt, fmt)`.

Ensures: `formatting()== fmt`.

```
    constexpr path_view_component(span<const byte> b, zero_termination zt) noexcept;
```

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

Effects: As if `path_view_component(b.data(), b.size(), zt)`.

Ensures: `formatting()== format::binary_format`.

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

Constraints:

1. `It` satisfies `contiguous_iterator`.
2. `End` satisfies `sized_sentinel_for<It>`.
3. `is_same_v<iter_value_t<It>, X>` is true for `X` being 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 `contiguous_iterator`.
3. `End` models `sized_sentinel_for<It>`.

Effects: As if `path_view_component(to_address(begin), end - begin, zt, fmt)`.

Ensures: `formatting()== fmt`.

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

Constraints:

1. `It` satisfies `contiguous_iterator`.
2. `End` satisfies `sized_sentinel_for<It>`.
3. `is_same_v<iter_value_t<It>, byte>` is true.
4. `is_convertible_v<End, size_type>` is false.

5. If `zt` is `zero_terminated`, then `*e == 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 `contiguous_iterator`.
3. `End` models `sized_sentinel_for<It>`.

Effects: As if `path_view_component(to_address(begin), end - begin, zt)`.

Ensures: `formatting()== format::binary_format`.

```
    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;
```

Path view components are trivial types, and therefore have all all-trivial copy and move constructors and assignment, and destructor.

Modifiers [fs.path_view_component.modifiers]

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

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

Observers [fs.path_view_component.observers]

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

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

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

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

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

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

```
constexpr bool zero_terminated() const noexcept;
```

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

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

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

Complexity: `O(native_size())`.

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

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

Complexity: `O(native_size())`.

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

Returns: Let `s` refer to one element 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())`.

> [*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]

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

Returns: Let `s` refer to one element 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())`.

```
template<class T = typename path::value_type,
         class Deleter = default_deleter<T[]>,
         size_type InternalBufferSize = default_internal_buffer_size>
constexpr int compare(path_view_component p) const;
```

Constraints: `is_same_v<T, X>` is true for `X` being any one of: `char`, `wchar_t`, `char8_t`, `char16_t`, `byte`; `invocable<Deleter, T*>` is true.

Effects:

- If `T` is `byte`, the comparison of the two backing data ranges is implemented as if `memcmp()`.
- Otherwise the comparison is implemented as if:

```
path_view_component::c_str<T, Deleter, 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(native_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::c_str` 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]

```
template<class T = typename path::value_type,
         class Deleter = default_deleter<T[]>,
         size_type InternalBufferSize = default_internal_buffer_size>
constexpr int compare(const path& p) const;

template<class T = typename path::value_type,
         class Deleter = default_deleter<T[]>,
         size_type InternalBufferSize = default_internal_buffer_size,
         class CharT>
constexpr int compare(const basic_string<CharT>& s, format fmt = path::auto_format) const;

template<class T = typename path::value_type,
         class Deleter = default_deleter<T[]>,
         size_type InternalBufferSize = default_internal_buffer_size,
constexpr int compare(const CharT* s, format fmt = path::auto_format) const;

template<class T = typename path::value_type,
         class Deleter = default_deleter<T[]>,
         size_type InternalBufferSize = default_internal_buffer_size,
constexpr int compare(basic_string_view<CharT> s, format fmt = path::auto_format) const;

template<class T = typename path::value_type,
         class Deleter = default_deleter<T[]>,
         size_type InternalBufferSize = default_internal_buffer_size>
constexpr int compare(span<const byte> s) const;
```

Constraints: `is_same_v<T, X>` is true for `X` being any one of: `char`, `wchar_t`, `char8_t`, `char16_t`, `byte`; `invocable<Deleter, T*>` is true.

Effects: As if `compare<T, Deleter, InternalBufferSize>(path_view_component(...))`.

```
    template<class T = typename path::value_type,
             class Deleter = default_deleter<T[]>,
             size_type InternalBufferSize = default_internal_buffer_size>
    constexpr int compare(path_view_component p, const locale& loc) const;
```

Constraints: `is_same_v<T, X>` is true for `X` being any one of: `char`, `wchar_t`, `char8_t`, `char16_t`, `byte`; `invocable<Deleter, T*>` is true.

Effects: As if `compare<T, Deleter, InternalBufferSize>` but where the as-if `path_view_component::c_str` is constructed with an additional `loc` argument.

```
    template<class T = typename path::value_type,
             class Deleter = default_deleter<T[]>,
             size_type InternalBufferSize = default_internal_buffer_size>
    constexpr int compare(const path& p, const locale& loc) const;

    template<class T = typename path::value_type,
             class Deleter = default_deleter<T[]>,
             size_type InternalBufferSize = default_internal_buffer_size,
             class CharT>
    constexpr int compare(const basic_string<CharT>& s, const locale& loc,
                          format fmt = path::auto_format) const;

    template<class T = typename path::value_type,
             class Deleter = default_deleter<T[]>,
             size_type InternalBufferSize = default_internal_buffer_size,
    constexpr int compare(const CharT* s, const locale& loc, format fmt = path::auto_format) const;

    template<class T = typename path::value_type,
             class Deleter = default_deleter<T[]>,
             size_type InternalBufferSize = default_internal_buffer_size,
    constexpr int compare(basic_string_view<CharT> s, const locale& loc, format fmt = path::
        auto_format) const;

    template<class T = typename path::value_type,
             class Deleter = default_deleter<T[]>,
             size_type InternalBufferSize = default_internal_buffer_size>
    constexpr int compare(span<const byte> s) const;
```

Constraints: `is_same_v<T, X>` is true for `X` being any one of: `char`, `wchar_t`, `char8_t`, `char16_t`, `byte`; `invocable<Deleter, T*>` is true.

Effects: As if `compare<T, Deleter, InternalBufferSize>(path_view_component(..., loc))`.

Struct `path_view_component::c_str` [fs.path_view_component.c_str]

```
namespace std::filesystem {
  template<class T = typename path::value_type,
           class Deleter = default_deleter<T[]>,
           size_type InternalBufferSize = path_view_component::default_internal_buffer_size>
```

```
  struct path_view_component::c_str {
    using value_type = T;
    using deleter_type = Deleter;
    static constexpr size_t internal_buffer_size = /* actual size of internal buffer used, which may
        differ from InternalBufferSize */;

    // Suggested same layout as span<const value_type> would have.
    const value_type* buffer{nullptr};
    size_type length{0};

  public:
    // constructors and destructor
    c_str() = default;
    ~c_str();

    template<class U, class V>
    constexpr c_str(path_view_component v, path_view_component::zero_termination zero_termination,
                    const locale& loc, U&& allocate, V&& deleter = deleter_type());

    template<class U, class V>
    constexpr c_str(path_view_component v, path_view_component::zero_termination zero_termination,
                    U&& allocate, V&& deleter = deleter_type());

    constexpr c_str(path_view_component v, path_view_component::zero_termination zero_termination);

    c_str(const c_str&) = delete;
    c_str(c_str&& o) noexcept;

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

  private:
    bool _call_deleter{false};                  // exposition only
    deleter_type _deleter;                      // exposition only
    value_type _buffer[internal_buffer_size]{}; // exposition only

    /* 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: `is_same_v<T, X>` is true for `X` being any one of: `char`, `wchar_t`, `char8_t`, `char16_t`, `byte`; `invocable<Deleter, T*>` is true.

Struct `path_view_component::c_str` 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::c_str` 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 `c_str` therefore does not introduce the usual concerns. 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 `c_str` 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 struct `path_view_component::c_str` nor `path`. For example, if the backing data is 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::c_str`.

[*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 supply the standard SNIA NVMe key-value API for storage devices. 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 `c_str` byte passthrough described below. – end note]

After construction, an object of struct `path_view_component::c_str` will have members `buffer` and `length`: `buffer` will point at an optionally zero terminated array of `value_type` of length `length`, which excludes any zero termination. As an example of usage with POSIX `open()`, which consumes a zero-terminated `const path::value_type*` i.e. `const char *`:

```
int open_file(path_view path)
{
  /* This function does not support binary key input */
  if(path.formatting() == path_view::format::binary_format)
  {
    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.buffer 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 c_str's internal buffer isn't big enough.
  */
  path_view::c_str<> zpath(path, path_view::zero_terminated);
  return ::open(zpath.buffer, 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. – end note]

```
~c_str();
```

Effects: If during construction a dynamic memory allocation was required (`_call_deleter == true`), that is freed using the `deleter_type` instance which was supplied during construction.

```
template<class U, class V>
constexpr c_str(path_view_component v, path_view_component::zero_termination zero_termination,
                const locale& loc, U&& allocate, V&& deleter = deleter_type());
```

Constraints: `invocable<U, size_t>` is true, and returns a type convertible to `value_type*`; `convertible_to<V, deleter_type>` is true.

Effects: The member variables `buffer` and `length` are set as follows:

- If `value_type` is `byte`, `length` is set to `v.native_size()`. If `zero_termination` is `zero_terminated` and `v.zero_terminated()` is false:
  - If `length < internal_buffer_size - 1`:
    - `buffer` is set to `_buffer`, the bytes of the backing data are copied into `buffer`, and a zero valued byte is appended.

    else:
    - `U(length + 1)` is invoked to set `buffer`, the bytes of the backing data are copied into `buffer`, and a zero valued byte is appended. `_call_deleter` is set to true.

  else:
  - `buffer` is set to the backing data.
- If the backing data is `byte` and `value_type` is not `byte`, `length` is set to `v.native_size()/sizeof(value_type)`. If `zero_termination` is `zero_terminated`, and either `(v.native_size()+v.zero_terminated())!= (length + 1)* sizeof(value_type)` is true or `v.zero_terminated()` is false:
  - If `length < internal_buffer_size - 1`:
    - `buffer` is set to `_buffer`, the bytes of the backing data are copied into `buffer`, and a zero valued `value_type` is appended.

    else:
    - `U(length + 1)` is invoked to set `buffer`, the bytes of the backing data are copied into `buffer`, and a zero valued `value_type` is appended. `_call_deleter` is set to true.

  else:

– `buffer` is set to the backing data.

[*Note:* The `(v.native_size()+ v.zero_terminated())!= (length + 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`. – 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), `length` is set to `v.native_size()`. If `zero_termination` is `zero_terminated` and `v.zero_terminated()` is false, 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 `length < internal_buffer_size - 1`:

    * `buffer` is set to `_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:

    * `U(length + 1)` is invoked to set `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. `_call_deleter` is set to true.

  else:

  – `buffer` is set to 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`. `buffer` shall point to that reencoded path, and `length` shall be the number of elements output, excluding any zero termination appended. If `U(...)` is invoked to dynamically allocate storage to store the reencoded path, `_call_deleter` is set to true. 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, we cannot use `loc` directly. Rather, we use `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]

```
template<class U, class V>
constexpr c_str(path_view_component v, path_view_component::zero_termination zero_termination,
                U&& allocate, V&& deleter = deleter_type());
```

Constraints: `invocable<U, size_t>` is true, and returns a type convertible to `value_type*`; `convertible_to<V, deleter_type>` is true.

Effects: The member variables `buffer` and `length` are set as for the preceding overload, using implementation defined mechanisms for performing reencodings. Implementations are required to raise 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.

```
constexpr c_str(path_view_component v, path_view_component::zero_termination zero_termination);
```

Effects: The member variables `buffer` and `length` are set as for the preceding overload, with `allocate` being an as if equivalent to `[](size_type length)return static_cast<value_type*>(::operator new [](length * sizeof(value_type))); ;`.

[*Note:* Some have asked about the 'weird' allocation and deallocation mechanism. The reason why is that we wish implementations to have the freedom to use a system API which dynamically allocates using a platform specific allocation the buffer it reencodes into, and set `buffer` to that allocation directly if the implementation then defines `default_deleter<T>` to be something able to deallocate such platform specific allocations. – end note]

Non-member comparison functions [fs.path_view_component.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;
```

Effects: If the backing bytes are of different encoding, the path view components are considered unequal. If the native sizes are unequal, the path view components are considered unequal. Otherwise as if `memcmp()` is used to compare the backing bytes of both path view components for equality.

```
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==(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!=(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!=(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;
```

Effects: Comparing for equality or inequality string literals, string views, byte literals or byte views, against a path view component is ill formed. 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. Even this paper's author 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]

```
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>& s, path_view_component v);
```

Effects: Equivalent to:

```
template<class charT, class traits>
basic_ostream<charT, traits>& operator<<(basic_ostream<charT, traits>& s, path_view_component v)
{
  return visit([&](auto sv) -> basic_ostream<charT, traits>& {
    if constexpr(same_as<remove_cvref_t<decltype(sv)>, 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::c_str<charT> zbuff(v, path_view_component::not_zero_terminated);
      return quoted(s.write(zbuff.buffer, zbuff.length));
    }
  });
}
```

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, and also has additional member functions implementing corresponding functionality from `path`.

`path_view` is required to be trivially copyable.

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

```
namespace std::filesystem {
  class path_view : public path_view_component {
  public:
    using const_iterator = /* implementation defined */;
    using iterator = /* implementation defined */;
    using reverse_iterator = /* implementation defined */;
    using const_reverse_iterator = /* implementation defined */;
    using difference_type = /* implementation defined */;

  public:
    // Constructors and destructors
    constexpr path_view() = default;

    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, zero_termination zt,
                        format fmt = path::auto_format) noexcept;
    constexpr path_view(const byte* b, size_type l, 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, zero_termination zt,
                        format fmt = path::auto_format) noexcept;
    constexpr path_view(span<const byte> b, zero_termination zt) noexcept;

    template<class It, class End>
    constexpr path_view(It b, End e, zero_termination zt,
                        format fmt = path::auto_format) noexcept;
    template<class It, class End>
    constexpr path_view(It b, End e, 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;

    // 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;

    constexpr path_view root_name() const noexcept;
    constexpr path_view root_directory() const noexcept;
    constexpr path_view root_path() const noexcept;
    constexpr path_view relative_path() const noexcept;
    constexpr path_view parent_path() const noexcept;
    constexpr path_view_component filename() const noexcept;
    constexpr path_view remove_filename() const noexcept;

    // Iteration
    constexpr const_iterator cbegin() const noexcept;
    constexpr const_iterator begin() const noexcept;
    constexpr iterator begin() noexcept;
    constexpr const_iterator cend() const noexcept;
    constexpr const_iterator end() const noexcept;
    constexpr iterator end() noexcept;

    // Comparison
    using path_view_component::compare;
    template<class T = typename path::value_type,
             class Deleter = default_deleter<T[]>,
             size_type InternalBufferSize = default_internal_buffer_size>
    constexpr int compare(path_view p) const;
    template<class T = typename path::value_type,
             class Deleter = default_deleter<T[]>,
             size_type InternalBufferSize = default_internal_buffer_size>
    constexpr int compare(path_view p, const locale &loc) const;

    // Conversion
    template<class T = typename path::value_type,
             class Deleter = default_deleter<T[]>,
             size_type InternalBufferSize = default_internal_buffer_size>
    struct c_str;
  };
}
```

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:* 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]

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

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

Complexity: `O(native_size())`.

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

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

Complexity: `O(native_size())`.

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

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

Complexity: `O(native_size())`.

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

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

Complexity: `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 `root_path()` until the end of the view, which may be an empty view.

Complexity: `O(native_size())`.

```
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())`.

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

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

Complexity: `O(native_size())`.

```
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())`.

> [*Note:* Comparisons and conversion are identical to the path view component constructors, and are not repeated here for brevity. – end note]

In 29.11.9.1 [fs.enum.path.format] paragraph 1:

\+ `binary_format` The binary pathname format.

3 Acknowledgements

4 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