# P3548: P1030 `std::filesystem::path_view` forward progress options


| | |
|---|---|
| Document #: | P3548 |
| Date: | 2025-01-13 |
| Project: | Programming Language C++<br>Library Evolution Working Group |


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


In the Wroclaw meeting, LEWG spent a full session with Victor presenting [P2645] *`path_view`: a design that took a wrong turn*, and then with me presenting a rebuttal to that paper. The lively discussion meant we ran out of time to poll the room for direction.

Victor's main concerns in the R1 of his paper were as follows:

1. Support for 'ANSI encoding' on Microsoft Windows (i.e. `char` input support).
2. Lack of reference implementation of filesystem free function overloads.
3. Lack of use of `render_null_terminated()`.
4. Performance will be slower not faster.

I believe I successfully convinced the room that concerns 2-4 are not an issue for standardisation, so this paper omits further mention of those[1].

That leaves concern 1, which Victor has confirmed to me by private correspondence, is what he considers the most important objection.

I informally polled a number of long standing LEWG members regarding point 1. I found three main camps.

The first camp takes the view that LEWG has seen six revisions of this paper (R7 is the current, it was not seen in the Wroclaw meeting). It is well understood by LEWG. LEWG knows what it is standardising, including the `char` input on Microsoft Windows, and it is happy with the current constructor set and the semantics being standardised.

The second camp agrees that `char` input for filesystem paths is problematic on Microsoft Windows only, and therefore it should be removed on Microsoft Windows only as the native filesystem path representation is `wchar_t`. This would mean that `char` string literals would cease to compile when on Microsoft Windows only, but would compile everywhere else.

The third camp feels that `char` input is problematic in general, and therefore `char` input should be removed on **all** platforms, thus forcing developers to explicitly declare a UTF-8 encoding using `char8_t` input. When it was pointed out that this could if strictly interpreted prevent a `path_view`

[1] Though see the bonus section below for more on performance.

being taken on a `path` on POSIX (because `path`'s underlying string representation is `char` on POSIX), members of this camp either switched to the first camp, wanted `path_view` input to be special cased, or wanted `path` to be deprecated and replaced entirely with an encoding aware replacement i.e. that a `path2` become original encoding aware like `path_view` is.

*Everybody* informally polled asked for a 'Tony table' mapping out the choices before LEWG with the consequences of each written out in clear, simple terms. That is this paper.

# 1 Reminder

As Steve Downey likes to say in every LEWG discussion of filesystem paths (because people forget every meeting), `std::filesystem::path` may look to most people as a kind of unicode string. In fact, they can be arbitrary binary bits so long as no codepoint is the path separator or null, and the filesystem or kernel may – or may not – apply various non one-one mappings between the path requested and the inode opened. These can include arbitrary unicode normalisations, which when acting on binary input can produce some interesting equivalences between different bitstring inputs!

Moreover, the C++ standard library is a *filesystem path taker*. It must accept arbitrary filesystem paths supplied as a bitstring by others, and correctly open the inode corresponding to that path as defined by the current configuration of the OS kernel and filesystem. **No matter what that bitstring contains!**.

This has a tension with the 'correct' rendering of filesystem paths to humans. The OS kernel and filesystem and standard library can be made so they don't 'futz' with the bitstring in breaking ways **so long** as no round trip via a human rendering occurs. If you want bitstring => human rendering => bitstring to result in a path which opens the same inode as the original path, this gets **much** harder. The problem is that you don't always know if human compatible rendering is needed e.g. if your program is currently writing to `cout`, if that is to TTY you probably want human compatible rendering. If `cout` currently points into a file, *maybe* you want human compatible rendering, but maybe you don't.

Current `std::filesystem::path` does a reasonable job of taking third party supplied paths and probably opening the inode intended *so long* as you don't serialise and deserialise the path. If you do, half the time it will work, half the time it won't, depending on how the result is used, as Victor's [P2319] *Prevent path presentation problems* tries to address, which was resolved by LEWG by having two member functions so the user can indicate *rendering intent* and thus avoid the 'half the time it's wrong' problem.

`path_view` adds explicit support for bitstrings (i.e. don't touch these bits!) as the backing representation, however `path` also implicitly supports those, just with the codepoints of the path separator and null excluded. Thus in the current `path_view` proposal the path => human rendering => path problem is essentially the same with `path_view` as it is with `path`, except that we know at the time of rendering what the original source encoding is with path view, which we do not with paths.

## 2 Tony table to aid decision taking

It is suggested that LEWG poll each of these, removing the one with the fewest votes each round, until a decision is taken. For the record, the author's opinion is option 3 which is to change nothing.

### 2.1 Remove the `char` backing representation on platforms where the native filesystem encoding is not in `char`

Pros:

- We do not add to the mess which is narrow system encoding to filesystem path encoding on Microsoft Windows.

Cons:

- C++ code on non-Microsoft Windows platforms which use `char` input to `path_view` would no longer compile on Microsoft Windows, thus losing portability. Additionally, `char` input to `path` would continue to compile on Microsoft Windows, so there would be dissimilarity.

Remarks:

- On Microsoft Windows, `filesystem::path`'s constructor for `char` input more or less does what `CreateFileA()` does such that later calls of `CreateFileW()` with the `wchar_t` native encoding of `filesystem::path` probably will open the same inode[2].

### 2.2 Remove the `char` backing representation on all platforms

Pros:

- Narrow system encoding to filesystem path encoding is not entirely trouble free on platforms other than Microsoft Windows either. The problem is that `char`'s meaning is overloaded: it could mean binary bits, it could mean ASCII, it could mean UTF-8, it could mean EBCDIC, or any arbitrary narrow system encoding. By preventing `char` input entirely, we would be consistent about path views across platforms.

Cons:

- `char` input to `path` would work as now, but `char` input to `path_view` would not compile. This might be felt by the userbase to be surprising, especially if `path` input to a `path_view` works when the underlying value type for `path` is `char`.

### 2.3 Leave the existing allowed backing representations of `path_view` as is, with the path view formatter doing exactly what the path formatter does

Pros:

[2] I say *probably*, because the narrow system encoding is a runtime possibly per-thread configurable setting which can vary at point of use.

- We do in `path_view` exactly what `path` does, so everything encoding related is equally as messy (or not) for path views as for path. This is consistent.

Cons:

- This adds to historical mistakes and does nothing to not keep furthering them.

Remarks:

- Microsoft created the mess which is narrow system encoding to filesystem path encoding on Microsoft Windows. This author believes it is on Microsoft to sort it out and WG21 should not get involved. I would point out that Microsoft's long term plan to turn the narrow system encoding to always UTF-8 appears to be on track, so in a decade or so this problem should be solved for newly compiled code.

### 2.4 Leave the existing allowed backing representations of `path_view` as is, omitting the path view formatter in C++ 26 until C++ 29

Pros:

- Because `path_view` retains awareness of the source encoding whereas `path` erases that knowledge, it could be possible to design a path view formatter which is superior to `path`'s formatter in terms of human rendering fidelity in some corner cases.

Cons:

- This would break one-one equivalence between formatting a `path` and `path_view`. This author is unsure if the value added would be worth the additional surprise factor and cognitive load.

Remarks:

- `path`'s formatter did not ship with `format` in C++ 20, but was deferred until C++ 26. There is precedent for doing the same with the path view formatter.

## 3 Bonus section: Performance Benchmark

My reading of [P2645] is that it is claimed `path_view` is slower than either `path` or a native string or some other way of doing paths (exactly which isn't entirely clear to me from that paper).

I've generally steered away from presenting benchmarks in my standards papers because of the 'how long is *that* piece of string?' problem i.e. for any benchmark one person can construct, another can usually construct a benchmark which shows the opposite results, and then entire days can be expended on people arguing about what `N + 1` actual or theoretical benchmarks might mean. For the purposes of engineering standards, benchmarks are therefore an excellent way of wasting committee time.

Still, I felt that paper's claims about performance needed rebutting. I made a recursive directory iterator in LLFIO which counts how many bytes of content there are in a directory tree of 128

million items. I made a **single line** change between the two benchmarks which is a 'dumb' swap of `path_view` for `path`:

Here we construct a `filesystem::path` from `entry.leafname` which is a `path_view`:

```
    for(const llfio::directory_entry &entry : contents)
    {
      // This line is the only difference
      fs::path leafname(entry.leafname.path());
      if(-1 == fstatat(dirh.native_handle().fd, leafname.c_str(), &s,
                       AT_SYMLINK_NOFOLLOW))
```

Here we render a null terminated path from `entry.leafname`:

```
    for(const llfio::directory_entry &entry : contents)
    {
      // This line is the only difference
      auto leafname = entry.leafname.render_null_terminated();
      if(-1 == fstatat(dirh.native_handle().fd, leafname.c_str(), &s,
                       AT_SYMLINK_NOFOLLOW))
```

This is a unrealistically unfair benchmark to `path_view` because the leafnames are very short, so the `malloc` behind `path` would use a fast pool of short allocations. The memory copying into the internal string would also be quick. Still, the `path` edition is approximately **20% slower** than the `path_view` edition.

To get a less unfair benchmark which uses longer paths which would impact `path` more, and that accounts for that the skilled developer wouldn't write identical code patterns for path views as for paths, I then made an additional benchmark which appends the leafname onto the absolute path of the containing directory and calls `lstat()` on the full length path. The `path` edition with these longer filesystem paths becomes **65% slower** than the short filesystem path benchmark above, which makes sense because now `malloc` is genuinely allocating regions and transcoding and copying the path into the internal string takes real time.

The `path_view` edition with these longer filesystem paths uses different code because we now have path views and rendered path available to the developer, so it would be expected that a skilled developer would make use of these new facilities and write more suitable code for path views. How much faster is the result?

1. `path` with short filesystem paths: 123.3 seconds.
2. `path_view` with short filesystem paths straight swap with `path`: 102.8 seconds.
3. `path` with longer filesystem paths: 206.5 seconds.
4. `path_view` with longer filesystem paths written to take advantage of path views: **18.77 seconds** (91% faster).

Once again, I would absolutely ignore any claims about benchmarks or performance, they're meaningless. But consider [P2645]'s performance claims rebutted.

The benchmark can be found at `https://github.com/ned14/llfio/blob/develop/example/wg21_path_view_benchmark.cpp`. You will almost certainly find it performs very differently on your

hardware. I ran mine on a thirty-two core Threadripper Pro with 128Gb of RAM running Ubuntu 24.04 x64 on an XFS filing system. I dropped filesystem caches before each run to make them run from cold cache, so your physical storage will be important: mine is two Samsung PM9A1 SSDs configured in RAID0 with XFS aware of the RAID0 configuration (and therefore optimising for it).

# 4 References


[P1030] Douglas, Niall
*std::filesystem::path_view*
https://wg21.link/P1030

[P2319] Zverovich, Victor
*Prevent path presentation problems*
https://wg21.link/P2319

[P2645] Zverovich, Victor
*path_view: a design that took a wrong turn*
https://wg21.link/P2645