# P1030R2: `std::filesystem::path_view`

| | |
|---|---|
| Document #: | P1030R2 |
| Date: | 2019-06-10 |
| Project: | Programming Language C++<br>Library Evolution Working Group |
| Reply-to: | Niall Douglas<br><s_sourceforge@nedprod.com> |

A proposal 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. Requires [P0482] *char8_t: A type for UTF-8 characters and strings.*

A mostly-conforming[1] 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 R1:
- Freshened stale URLs to AFIO, which was renamed to LLFIO a year ago.

## Contents

[1]The differences are mainly due to lack of `char8_t`.

## 1 Introduction

In the current C++ standard, the canonical way for supplying filesystem paths to C++ functions which consume file system paths is `std::filesystem::path`. This wraps up `std::filesystem::path::string_type` (= `std::basic_string<Char>`) with a platform specific choice of `Char` (currently Microsoft Windows uses `Char` = `wchar_t`, everything else uses `Char` = `char`) with iterators and member functions which parse the string according to the path delimiters for that platform. For example `std::filesystem::path` on Microsoft Windows might parse this string:

`C:\Windows\System32\notepad.exe`

into:

- `root_name()` = "`C:`"
- `root_directory()` = "`\`"
- `root_path()` = "`C:\`"
- `relative_path()` = "`Windows\System32\notepad.exe`"
- `parent_path()` = "`C:\Windows\System32`"
- `filename()` = "`notepad.exe`"
- `stem()` = "`notepad`"
- `extension()` = "`.exe`"
- `*begin()` = "`C:`"
- `*++begin()` = "`/`" (note the forward, not backward, slash. This is considered to be the name of the root directory)
- `*++++begin()` = "`Windows`"
- `*++++++begin()` = "`System32`" (note no intervening slash)

For every one of these decompositions, a new `path` is returned, which means a new underlying `std::basic_string<Char>`, which means a new memory allocation. In code which performs a lot of path traversal and decomposition, these memory allocations, and the copying of fragments of path around, can start to add up. For example, in [P1031] *Low level file i/o library*, a directory enumeration costs around 250 *nanoseconds* per entry amortised. Each path construction might cost that again. Therefore, for each item enumerated, one *halves* the directory enumeration performance solely due to the choice of `path`, which is why P1031 uses `path_view` instead, and thus can enumerate four million directory items per second, which makes handling ten million item plus directories tractable.

There is also a negative effect on CPU caches of copying around path strings. Paths are increasingly reaching hundred of bytes, as anyone running into the 260 path character limit on Microsoft Windows can testify[2]. Every time one copies a path, one is evicting potentially useful data from the CPU

[2]You can now build your Windows application with this limit removed for your program.

caches, which need not be evicted if one did not copy paths.

Enter thus the proposed `std::filesystem::path_view`, which is a lightweight reference to part, or all of, a source of filesystem path data. It provides most of the same member functions as `std::filesystem::path`, operating by constant and often constexpr reference upon some character source which is in the format of the local platform's file system path, or a generic path, same as with `std::filesystem::path`. It is intended that for most functions currently accepting a `std::filesystem::path`, they can now accept a `std::filesystem::path_view` instead with minor to none refactoring of implementation.

## 2 Impact on the Standard

The proposed library is a pure-library solution.

## 3 Proposed Design

Much of the proposed path view is unsurprising, with a large subset of `std::filesystem::path`'s observers and modifiers replicated. Constexpr abounds, and the path view is trivially copyable and is thus suitable for passing around by value.

```
class path_view
{
public:
  // Const iterator, returns path views of each path section.
  class const_iterator;
  // Iterator, aliases const iterator.
  class iterator;
  // Const reverse iterator
  class const_reverse_iterator;
  // Reverse iterator
  class reverse_iterator;
  // Size type
  using size_type = std::size_t;
  // Difference type
  using difference_type = std::ptrdiff_t;

public:

  // Constructs an empty path view
  constexpr path_view();
  ~path_view() = default;

  // Implicitly constructs a path view from a path.
  // The input path MUST continue to exist for this view to be valid.
  path_view(const filesystem::path &v) noexcept;


  // Implicitly constructs a path view from an unencoded byte string.
  // The input string MUST continue to exist for this view to be valid.
```

```
path_view(const std::basic_string<std::byte> &v) noexcept;

// Implicitly constructs a path view from a UTF-8 string.
// The input string MUST continue to exist for this view to be valid.
path_view(const std::basic_string<std::char8_t> &v) noexcept;

// Implicitly constructs a path view from a UTF-16 string.
// The input string MUST continue to exist for this view to be valid.
path_view(const std::basic_string<std::char16_t> &v) noexcept;

// Implicitly constructs a path view from a UTF-32 string.
// The input string MUST continue to exist for this view to be valid.
path_view(const std::basic_string<std::char32_t> &v) noexcept;


// Constructs a path view from a lengthed byte sequence.
// \warning The character after the end of the view must be legal to read.
constexpr path_view(const std::byte *v, size_t len) noexcept;

// Implicitly constructs a path view from a zero terminated UTF-8 character sequence.
// The input string MUST continue to exist for this view to be valid.
constexpr path_view(const char8_t *v) noexcept;

// Constructs a path view from a lengthed UTF-8 character sequence.
// \warning The character after the end of the view must be legal to read.
constexpr path_view(const char8_t *v, size_t len) noexcept;

// Implicitly constructs a path view from a zero terminated UTF-16 character sequence.
// The input string MUST continue to exist for this view to be valid.
constexpr path_view(const char16_t *v) noexcept;

// Constructs a path view from a lengthed UTF-16 character sequence.
// \warning The character after the end of the view must be legal to read.
constexpr path_view(const char16_t *v, size_t len) noexcept;

// Implicitly constructs a path view from a zero terminated UTF-32 character sequence.
// The input string MUST continue to exist for this view to be valid.
constexpr path_view(const char32_t *v) noexcept;

// Constructs a path view from a lengthed UTF-32 character sequence.
// \warning The character after the end of the view must be legal to read.
constexpr path_view(const char32_t *v, size_t len) noexcept;


/* Implicitly constructs a path view from a UTF-8 string view.
\warning The character after the end of the view must be legal to read.
*/
constexpr path_view(basic_string_view<char8_t> v) noexcept;

/* Implicitly constructs a path view from a UTF-16 string view.
\warning The character after the end of the view must be legal to read.
*/
constexpr path_view(basic_string_view<char16_t> v) noexcept;

/* Implicitly constructs a path view from a UTF-32 string view.
\warning The character after the end of the view must be legal to read.
```

```
*/
constexpr path_view(basic_string_view<char32_t> v) noexcept;

// Default copy constructor
path_view(const path_view &) = default;
// Default move constructor
path_view(path_view &&o) noexcept = default;
// Default copy assignment
path_view &operator=(const path_view &p) = default;
// Default move assignment
path_view &operator=(path_view &&p) noexcept = default;

// Swap the view with another
constexpr void swap(path_view &o) noexcept;

// True if empty
constexpr bool empty() const noexcept;

// Exactly the same as for filesystem::path
constexpr bool has_root_path() const noexcept;
constexpr bool has_root_name() const noexcept;
constexpr bool has_root_directory() const noexcept;
constexpr bool has_relative_path() const noexcept;
constexpr bool has_parent_path() const noexcept;
constexpr bool has_filename() const noexcept;
constexpr bool has_stem() const noexcept;
constexpr bool has_extension() const noexcept;
constexpr bool is_absolute() const noexcept;
constexpr bool is_relative() const noexcept;

// Adjusts the end of this view to match the final separator, same as filesystem::path
constexpr void remove_filename() noexcept;

// Returns the size of the view in characters, same as filesystem::path::native().size()
constexpr size_t native_size() const noexcept;

// Exactly the same as for filesystem::path, but returning a slice of this view
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 filename() const noexcept;
constexpr path_view stem() const noexcept;
constexpr path_view extension() const noexcept;

// Return the path view as a filesystem path.
filesystem::path path() const;

/*! Compares the two string views via the view's ‘compare()‘ which in turn calls
‘traits::compare()‘. Be aware that if the path views do not view the same underlying
representation, a UTF based comparison will occur rather than a ‘memcmp()‘ of
the raw bytes.
*/
constexpr int compare(const path_view &p) const noexcept;

```

```
  // iterator is the same as const_iterator
  constexpr iterator begin() const;
  constexpr iterator end() const;

  // Instantiate from a ‘path_view‘ to get a zero terminated path suitable for feeding to the kernel
  struct c_str;
  friend struct c_str;
};

// Usual free comparison functions
inline constexpr bool operator==(path_view x, path_view y) noexcept;
inline constexpr bool operator!=(path_view x, path_view y) noexcept;
inline constexpr bool operator<(path_view x, path_view y) noexcept;
inline constexpr bool operator>(path_view x, path_view y) noexcept;
inline constexpr bool operator<=(path_view x, path_view y) noexcept;
inline constexpr bool operator>=(path_view x, path_view y) noexcept;
inline std::ostream &operator<<(std::ostream &s, const path_view &v);
```

There is a child helper struct which takes in a path view, and decides whether the path view needs to be copied onto the stack due to needing zero termination and/or UTF conversion, or whether the original collection of bytes can be passed through without copying.

```
  struct c_str
  {
    // Maximum stack buffer size on this platform
#ifdef _WIN32
    static constexpr size_t stack_buffer_size = 32768;
#elif defined(PATH_MAX)
    static constexpr size_t stack_buffer_size = PATH_MAX;
#else
    static constexpr size_t stack_buffer_size = 1024;
#endif

    // Number of characters, excluding zero terminating char, at buffer
    // Some platforms e.g. Windows can take sized input path buffers, and thus
    // we can avoid a memory copy to implement null termination on those platforms.
    uint16_t length{0};

    // A pointer to a native platform format file system path
    const filesystem::path::value_type *buffer{nullptr};

    c_str(const path_view &view) noexcept;
    ~c_str();
    c_str(const c_str &) = delete;
    c_str(c_str &&) = delete;
    c_str &operator=(const c_str &) = delete;
    c_str &operator=(c_str &&) = delete;

  private:
    // Flag indicating if buffer was malloced
    bool _buffer_needs_freeing;

    // Compilers don’t actually allocate this on the stack if it can be
    // statically proven to never be used. They optimise it out entirely.
    filesystem::path::value_type _buffer[stack_buffer_size]{};
  };
```

The use idiom would be as follows:

```
int open_file(path_view path)
{
  // I am on POSIX which requires zero terminated char filesystem paths.
  // So here if the character after the end of the view is zero, and the view
  // refers to char* data, we can use it directly without memory copying.
  path_view::c_str p(path);
  return ::open(p.buffer, O_RDONLY);
}
```

You will surely note the requirement that the character after the path view is legal to read. In this regard, path views are different to string views.

## 4 Design decisions, guidelines and rationale

There are a number of non-obvious design decisions in the proposed path view object. These decisions were taken after a great deal of empirical trial and error with 'more obvious' designs, where those designs were found wanting in various ways. The author believes that the current set of tradeoffs is close to the ideal set.

The design imperatives for an allocating `std::filesystem::path` are not those for a non-allocating `std::filesystem::path_view`. A 'handy feature' of an allocating path object is that it must always copy its input into its allocation. If it is allocating memory and copying in any case, performing an implicit conversion of a native narrow input encoding to say a native wide encoding seems like a reasonable design choice, given the relative cost of the other overheads.

In the case of a path view however, we are trying very hard to not copy memory. If the local platform uses the same narrow or wide input encoding as the source backing the view, and the path view is already terminated by a null character where that is relevant on the local platform, no copying is required. The original is used unmodified, bytes are passed through as-is. Only if necessary, a copy and/or conversion of the input onto the stack is performed into whatever format the local platform requires.

One might argue that in the case of `std::filesystem::path`, we might reuse the path across multiple calls, and thus the path view approach of just-in-time copying per syscall is wasteful on those platforms. However it is exceeding rare to open the same file more than once, and anyone caring strongly about performance will simply modify their source to use the same native encoding and null termination as the platform.

The next argument is usually one of the form that paths get commonly reused with just the leafname modified, and therefore path's approach is more efficient as only the leafname gets converted per iteration. I would counter that this proposed path view object comes from [P1031] *Low level file i/o library* where using absolute paths is bad form: you use a `path_handle` to indicate the base directory and supply a path view for the leafname – this is *far* more efficient than any absolute path based mechanism as it avoids the kernel having to traverse the filesystem hierarchy, typically taking a read lock on each inode in the absolute path.

## 4.1 Requiring legality of read of character after end of view

The reason for this is obvious: POSIX and Win32 syscalls consume zero terminated strings as path input, so we need to probe the character after the path view ends to see if it is zero, because if it is not, then we will need to copy the path view onto the stack in order to zero terminate it.

The question is whether this is dangerous or not. `string_view` does not do this, but then string views have a much wider set of use cases, including encapsulating a 4Kb page returned by `mmap()` where reading the byte immediately after the end of the view would mean a segmentation fault.

Path views do not suffer from that problem. One knows that they represent a path on the file system, and are very likely to be constructed from a source whose representation of a file system path will not vary by much. They are therefore highly unlikely to not be zero terminated *at some point* later on, as operations on path views only ever produce sub-views of some original path, and cannot escape the bounds of the original path. Path string literals are safe, by definition. Even a deserialised path from storage is highly likely to always be zero terminated. Because of the much more limited set of use cases for path views, I believe that this requirement is safe.

There is the separate argument that deviating requirements from `string_view` is unhelpful by confusing the user base, and will produce buggy code. Yet I cannot think of a single non-contrived use case where the legality of reading the character after the end of a path view is problematic, including tripping any static analysis tools or sanitisers. I welcome non-contrived examples of where this is dangerous.

I appreciate that if standardised, most in the C++ user base will not know of this requirement and will write code not taking this requirement into account. I would argue that it will be very unusual for the ignorant to become surprised by this requirement.

## 4.2 Fixed use of stack in `struct c_str`

Firstly, note that the compiler elides completely the fixed stack buffer for zero termination and UTF conversions caused by instantiating `struct c_str` if the compiler can prove that it will never be used. So if you supply native format, zero terminated input, to the path view constructor, the compiler should spot that the temporary stack buffer is never used, and thus eliminate it. This ought to be the case most of the time.

Secondly, the fixed stack buffer tends to get allocated just before a syscall, and released just after that syscall. Stack cache locality is therefore generally unaffected, and the fixed stack buffer does not remain allocated for long.

64Kb on Microsoft Windows systems may seem excessive, but it is highly unlikely to be a problem in practice as Windows has ample default stack address space reservations, and these allocations never recurse. Of the major POSIX implementations, Linux would potentially allocate 4Kb, MacOS 1Kb, FreeBSD 1Kb onto the stack as those are the settings for their `PATH_MAX` macros.

For those Linux implementations running on embedded systems where 4Kb stack allocations would be unwise, we do provide for the ability to internally use `malloc()` to create storage for the temporary buffer which is freed on `struct c_str`'s destruction. One could, of course, decide that on Microsoft Windows 8Kb of stack is enough, and paths larger than that go to `malloc()`. Again, I would stress that the programmer can be careful to never send a non-zero terminated string in as a path, and thus completely eliminate the use of temporary buffers on an embedded Linux solution. In any case, path views are considerably less heavy on free RAM than `std::filesystem::path`.

## 5 Technical specifications

No Technical Specifications are involved in this proposal.

## 6 Frequently asked questions

### 6.1 Where is the `char` input?

Unfortunately the encoding of `char` input is ambiguous, unlike `std::byte` (unencoded raw bits), or `char8_t` (UTF-8). We therefore do not accept `char` input, forcing the user to explicitly indicate the encoding of input.

### 6.2 Does this mean that all APIs consuming `std::filesystem::path` ought to now consume `std::filesystem::path_view` instead?

Most of the time, perhaps almost always, yes. `std::filesystem::path_view` implicitly constructs from explicitly encoded strings, paths and explicitly encoded string literals. Anywhere you are currently consumg `std::filesystem::path` as a parameter, you can start using `std::filesystem::path_view` instead if this proposal is approved. It would remain the case that where a function is returning a new path, `std::filesystem::path` is the correct choice. So inputs would be mostly path views, outputs would be paths.

This author has replaced paths with path views in an existing piece of complex path decomposition and recomposition, and apart from a few minor source code changes to fix lifetime issues, the code compiled and worked unchanged. Path views are mostly a drop-in replacement for paths, except for when one is creating wholly new paths.

Incidentally, performance of that code improved by approximately twenty fold (20x).

## 7 Acknowledgements

My thanks to Nicol Bolas, Bengt Gustafsson and Billy O'Neal for their feedback upon this proposal.

## 8 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