DDDDDR0 draft 1: `std::filesystem::path_view`

Document #:	DDDDDR0 draft 1
Date:	2018-04-19
Project:	Programming Language C++ Library Evolution Working Group

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

A proposal for a `std::filesystem::path_view`, a non-owning view of character sequences in the format of a local filesystem path.

A reference implementation of the proposed path view can be found at `https://ned14.github.io/afio/`. It has been found to work well on recent editions of GCC, clang and Microsoft Visual Studio, on x86, x64, ARM and AArch64.

Changes since draft 1:
- First edition.

Contents

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, so for example `std::filesystem::path` on Microsoft Windows might parse:

```
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 [PGGGG] *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 path decomposition, one halves the directory enumeration performance.

There is also a negative effect on CPU caches of copying around path strings. Paths are increasingly reaching 256 bytes, as anyone running into the 260 path character limit on Microsoft Windows can testify. Every time one copies a path, one is evicting potentially useful data from the CPU caches which need not be evicted if one did not copy paths.

Enter thus the proposed `std::filesystem::path_view`, which is to `std::filesystem::path` as `std::string_view` is to `std::string`. It provides most of the same member functions as `std::filesystem::path`, operating by constant reference upon some character source which is in the format of the local platform's file system path.

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.

One thing which is perhaps surprising is that the value type is *always* a `char`, not a `std::filesystem::path::value_type`. In other words, we always present a path view of *characters*, not wide characters, irrespective of local platform convention. This design choice will be explained later.

```
class path_view
{
public:
  // Character type
  using value_type = char;
  // Pointer type
  using pointer = char *;
  // Const pointer type
  using const_pointer = const char *;
  // Reference type
  using reference = char &;
  // Const reference type
  using const_reference = const char &;
  // Const iterator
  class const_iterator;
  // 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 UTF-8 path view from a string.
  // The input string MUST continue to exist for this view to be valid.
  path_view(const std::string &v) noexcept;

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

  // Constructs a UTF-8 path view from a lengthed `const char *`.
  // The input string MUST continue to exist for this view to be valid.
  constexpr path_view(const char *v, size_t len) noexcept;

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

#ifdef _WIN32
  // Implicitly constructs a UTF-16 path view from a string.
  // The input string MUST continue to exist for this view to be valid.
  path_view(const std::wstring &v) noexcept;

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

  // Constructs a UTF-16 path view from a lengthed `const wchar_t *`.
  // The input string MUST continue to exist for this view to be valid.
  constexpr path_view(const wchar_t *v, size_t len) noexcept;

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

  // 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
  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 path, performing UTF conversion to the local
  // filesystem::path::string_type if necessary
  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
  UTF, a temporary conversion of the non-UTF-8 view to UTF-8 view is performed on to
  the stack.
  */
  constexpr int compare(const path_view &p) const noexcept;
  constexpr int compare(const char *s) const noexcept;
  constexpr int compare(string_view str) const noexcept;
#ifdef _WIN32
  constexpr int compare(const wchar_t *s) const noexcept;
  constexpr int compare(wstring_view str) const noexcept;
#endif

  // 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
    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 UTF-8 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 the ideal set.

4.1 Ignore what character type `std::filesystem::path::value_type` is

The first non-obvious design choice is always presenting the view as byte characters, with a presumption of it being in UTF-8, not whatever `std::filesystem::path::value_type` is.

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 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, no conversion is required. The source backing is used unmodified. If the input is narrow but the platform is wide, `c_str` assumes that the input is UTF-8, and converts it to a UTF-16 representation on the stack which is passed to the syscall.

Note that the number of encoding conversions performed between the original input and the syscall remains unchanged. It is no more, and no less than before.

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 is wasteful. However it is exceeding rare to open the same file more than once, and anyone caring strongly about performance will simply modify their program to use the same native encoding 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 [PGGGG] *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.

The final argument is 'why interpret chars as UTF-8 when the normal path object does not?' That is a good question, which is answered next.

4.2 Why interpret chars as UTF-8 when `std::filesystem::path` interprets chars as 'the native narrow encoding'?

`std::filesystem` came originally from Boost.Filesystem, which in turn underwent three major revisions during the Boost peer review as it was such a lively debate. During those reviews, it was considered very important that paths were passed through, unmodified, to the system API. There are very good reasons for this, mainly that filesystems, for the most part, treat filenames as a bunch of bytes without interpreting them as anything. So any hard to avoid character reencoding could cause a path entered via copy-and-paste from the user to be unopenable, unless the bytes were passed through exactly.

This was and is a laudable aim, and it is preserved in this path view proposal. Unfortunately it has a most unfortunate side effect: on Microsoft Windows, `std::filesystem::path` when supplied with chars, is considered to be in *ANSI* encoding. This is because the char accepting syscalls on Microsoft Windows consume ANSI. On all other platforms, the chars are considered to be in UTF-8 encoding.

Boost.Filesystem was approved into Boost based on design priorities for Boost, which are not necessarily aligned with those of the C++ standard. In particular, ANSI Windows build compatibility is important for Boost, despite that no C++ project for Windows started in the last twenty years would configure an ANSI build[1]. If Boost.Filesystem chose to interpret char strings as UTF-8, that would utterly break ANSI builds of Boost, and the alternative of 'char strings are ANSI on ANSI builds, UTF-8 on Unicode builds' was not considered safe at the time. Hence the `std::filesystem::path` which entered the C++ 17 standard preserved this design choice of 'char strings mean ANSI on Windows, everywhere else UTF-8', despite it making much less sense outside of the Boost C++ Libraries.

On Microsoft Windows the native wide encoding is UTF-16, and modern C++ compilers gladly accept UTF-8 source code. This in turn means that `"UTF♠stringΩliteral"` makes a UTF-8 char string, and the L-prefixed `L"UTF♠stringΩliteral"` makes a UTF-16 `wchar_t` string.

This author can only speak from his own personal experience, but what he has found over many years of practice is that one ends up inevitably `#ifdef`-ing your `std::filesystem::path` based program code to use `L"UTF♠stringΩliteral"` when `_WIN32` and `_UNICODE` are macro defined, and otherwise uses `"UTF♠stringΩliteral"`. The reason is simple: the same string literal, with merely a `L` or not prefix, works identically on all platforms, no surprises, because we are writing string literals in some UTF-x format. Yet you end up spamming your program code with string literal wrapper macros as if we were still writing for MFC, and/or `#if defined(_WIN32) && defined(_UNICODE)` all over your code. I do not find this welcome.

I appreciate that [P0882] *User-defined Literals for std::filesystem::path* will take much of this sort of pain away, and some future Unicode string support for C++ from SG16 will no doubt do better

[1] Even Windows 95 supported Unicode builds, though they were buggy. Unicode builds have been the de facto choice from Windows 2000 onwards.

again. But as those are not yet approved additions to the standard, I propose that when supplied as a path string literal, and if and only if a conversion is needed, that we interpret chars as UTF-8 on all platforms rather than one behaviour on some platforms and another behaviour on other platforms. I suspect that if the Boost peer review were run without ANSI Windows build compatibility requirements in mind, reviewers would have also landed on UTF-8 being the correct interpretation of chars when supplied as a path.

4.3 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 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.

4.4 Fixed use of stack in `struct c_str`

Firstly, note that the compiler elides completely the fixed stack buffer for 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 99% 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.

Microsoft Windows systems are highly unlikely to need to worry about 64Kb being temporarily allocated on the stack as they have ample default stack address space reservations. 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. 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`.

4.5 Only native narrow encoded input is supported, except on Microsoft Windows where native wide encoded input is also supported

Unlike `std::filesystem::path` which accepts (i) native narrow encoding (ii) native wide encoding (iii) UTF-8 (iv) UTF-16 and (v) UTF-32 format input (all of which have an implied encoding conversion to internal native format, if necessary), path views need to discourage unnecessary reencoding, as due to it not being cached anywhere, it is inefficient to use an input format not equal to the native format.

To that end, path views accept the native narrow encoding only, except on Microsoft Windows where the *true*[2] native encoding is always wide since Microsoft Windows 2000.

Portable code will therefore be inefficient on Microsoft Windows, but that is no different to the present situation with `std::filesystem::path`. If, at some future point, Microsoft decides to convert their native encoding to UTF-8, we can remove the inefficiency on their platform.

5 Technical specifications

No Technical Specifications are involved in this proposal.

6 Frequently asked questions

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

Basically, yes. `std::filesystem::path_view` implicitly constructs from strings, paths and string literals. Anywhere you are currently using `std::filesystem::path`, you can start using `std::filesystem::path_view` instead if this proposal is approved.

[2] On Microsoft Windows, the ANSI APIs allocate a buffer and copy the narrow bytes into a wide byte form, then call the equivalent Unicode API.

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

Todo

8 References

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

[PGGGG] Douglas, Niall
Low level file i/o library
https://wg21.link/PGGGG