On Windows, Systems APIs, Encodings, and Pragmatism

Document #: P3670R0
Date: 2025-04-16
Programming Language C++
Audience: SG-16
Reply-to: Corentin Jabot <corentin.jabot@gmail.com>

Abstract

This paper proposes no change. Instead, it hopes to guide the design of future papers dealing with Unicode in system interfaces. This is very much an opinion piece, I hope you enjoy the read.

Characterizing the ecosystem

Windows deals in UTF-16

Most Windows APIs, including its filesystem APIs, are natively UTF-16 (specifically UTF-16LE). This has been the case since the release of Windows 2000, i.e., for the past 25 years.

Yet we keep seeing on the Internet and in committee discussions that Windows deals with UCS-2. This is incorrect, and has been incorrect for a very, very long time.

Windows can understand surrogate pairs and represent codepoints outside of the BMP. UCS-2 is dead.

What is correct to say is that not all Windows APIs accepting sequences of Unicode codepoints sanitize their inputs. This is most notably the case for paths:

The Documentation states:

> There is no need to perform any Unicode normalization on path and file name strings for use by the Windows file I/O API functions because the file system treats path and file names as an opaque sequence of WCHARs. Any normalization that your application requires should be performed with this in mind, external of any calls to related Windows file I/O API functions.

It should be noted that this quote non-withstanding, Microsoft does not do a particularly good job documenting which APIs accept arbitrary `wchar_t` sequences and which do not. For example, `CreateFileW` seems to sometimes, but not always, reject invalid UTF-16.

There is no validation in low-level APIs, such that creating Paths with embedded nulls with `NtCreateFile` is possible.

Learning these cool factoids should be the end of the story.

Alas, the prevailing wisdom in WG21, SG16, and most other language communities seems to be that because Windows can technically do all of these things, it behooves the standard library to support them.

Linux deals in UTF-8

The usual observation is that paths on Linux are just a bag of bytes. This observation is mostly accurate, albeit, for example, ZFS can be configured to do UTF normalization, and some filesystems can be configured to be case-insensitive.

But beyond the kernel, there is an expectation that paths appear in user-facing places such as graphical file explorers, the output of `ls`, the arguments to `find`, logs, source code etc.

Because not being able to render paths as text would be inconvenient, paths are UTF-8 encoded by convention. Files not created by end users are further likely to be restricted to the ASCII subset of UTF-8.

Non-Unicode platforms

Some platforms use non-Unicode and even non-ASCII encodings. But even on these platforms, it is fair to assume that the intent is for most paths to be human-readable text.

Most paths are text

The conclusion of all of that is that most paths are representable in Unicode. How much is most? It is hard to say, but probably more than 99.9Most users will never interact with ill-formed path names on any platform. There are none on my (Linux) system (some paths did contain emojis and CJK characters for some reason!).

What does that mean for C++?

Regardless of encodings, we have to admit the existence of invalid code unit sequences in paths. Malformed data, whether constructed by the user or by the filesystem, is a fact of life we can’t ignore.

We should actively not support embedded nulls

Embedded nulls are routinely a source of Bugs and CVEs, and there is no user-facing way to create a path with embedded nulls, so this is something we should actively not care about or, when practical, error on. Given the potential safety implication of letting these embedded nulls propagate in a program, I think the question is less whether they are worth supporting but whether they might be worth diagnosing early.

Provide limited support for non-well-formed Unicode paths on Windows

In general, there is no difficulty in supporting non-well-formed paths in places where we accept or return sequences of `wchar_t` and the internal storage is also a sequence of `wchar_t`, which is the case for `std::path` on windows.

But there are scenarios that we should actively not support:

- Converting a non-well-formed path to Unicode and back and expecting that to be a valid path still
- Transforming (for example, upper casing) a non-well-formed path to Unicode and expect that to work
- Printing out a non-well-formed path on a terminal and expecting that printed path to still be usable to address the file.

In general, trying to have byte-preserving round-tripping transformations on non-well-formed Unicode sequences adds orders of complexity.

Similarly, for a well-formed string in a non-Unicode encoding, we cannot guarantee round-tripping to be byte preserving (as conversion tables are not specified and could contain slight variations such that they would preserve semantics but not byte representation).

In fact, the only guarantee we can offer is that for a string that is valid Unicode, round-tripping conversions are codepoint-preserving.

The Rust approach

Because the question was asked in a previous SG-16 meeting, I figured it would be interesting to look at how Rust handles string. Rust strings are (valid) UTF-8 sequences on all platforms - and, unrelatedly, are not null-terminated.

But the `Path` type deals in `OSString`, where `OSString` stores a sequence of `u8` that is

- UTF-8 on Linux
- WTF-8 on Windows

This design allows Rust to use the same types with the same layouts and function across platforms while supporting arbitrary Windows paths that may not be valid Unicode.

This is arguably a more cohesive solution than `wchar_t` (which requires duplications of APIs). Note that Rust had the luxury of hindsight, and WTF-8 was only specified around 2014.

But in that scenario, `OSString`'s goal is more to be a portable alternative to `wchar_t` than to support round-tripping everywhere.

To use `OSString` with native Windows API, it is first converted to `OsStr` (to add a null terminator), and then to a sequence of `u16`.

On all platforms, `OSString` offers APIs to convert to UTF-8, either injecting replacement characters or returning an error on conversion failure.

`std::arguments`

This paper is a direct response to P3474R0 [4] (`std::arguments`), or rather the ensuing discussions, although it is something I have been itching to ink for a while.

The premise is simple: Let's expose `argv` to some globally accessible list of string-ish objects. But quid of the internal representation, encodings, conversions, and related APIs?

One observation made by the paper is that, on Windows, program arguments are UTF-16 encoded and `argv` is merely a lossy conversion provided either for conformance or convenience.

But maybe we want to support for `__wargv` too. And maybe we want to expose arguments as Unicode-encoded strings. And maybe support opening arbitrary paths that are not representable in Unicode.

It's a lot of conflicting requirements.

The current interface contains

```
string_view_type native() const;
const value_type* c_str() const;
std::string string() const;
std::wstring wstring() const;
std::u8string u8string() const;
std::u16string u16string() const;
std::u32string u32string() const;
// maybe
filesystem::path path() const;
```

Which is... a lot. This is by no means a criticism of the -otherwise excellent - paper; an interface with conflicting requirements is invariably going to lead to an increase in complexity. Nevertheless, adding so many ways to represent an argument does not do users or implementers any favors. It should be simple. Not that there are multiple problems that this interface is trying to address:

- The desire to be zero-copy while being able to produce null-terminated strings, despite the absence of a standard null-terminated `string_view` type (P3655R0 [1])
- The desire to support both `char` and `wchar` in the same type.
- The desire to offer support of all Unicode encodings on the premises of "someone might need it" or "why not?", even as the standard does not expose any generic conversion facilities.
- The desire to treat any argument as a valid path, even in the presence of lone surrogates on Windows.

Rust, again

In comparison, Rust has exactly two functions, one that returns a range of `String` (`std::u8string`) - and fails in the presence of not valid Unicode, and one that returns a range of OsString (either unvalidated UTF-8 or WTF-8). It's a much cleaner interface.

What are our options?

We could go the Rust route and embrace WTF-8. But it would be extremely inconsistent and uncompatible with existing code. It also incurs extra copies, which, while probably acceptable, is something C++ tries to avoid.

We should observe that wide arguments are specific to Windows. Indeed, the standard mandates that the "argv" argument to `main`, when it exists, is a `char**`. So one solution would be to provide `std::arguments` that only deal with `char` and, on Windows, `std::warguments`. In particular, we should not burden non-windows users and implementers with the thought of `wchar_t`.

```
struct argument {
    zstring_view str() const;
    u8string to_utf8() const;
};
struct wargument { // Windows specific
    zwstring_view str() const;
    u8string to_utf8()  const;
};
```

Which seems much simpler to use correctly. And removing the requirements that unpair surrogated and embedded nulls, we have more flexibility in terms of validation, preconditions, internal representation, etc.

Use the platform APIs, Luke

At the end of the day, the time of SG16, paper authors, and implementers is limited. Unicode support in the standard library is lackluster in that we do not have the most basic support of UTF-8.

In that context, worrying about the 0.01% use cases of lone surrogates in path names does seem like a counter-productive use of time. It's not serving users very well, and we should recognize that there is no obligation for the standard library to support every idiosyncrasy of every platform at the cost of additional efforts, more complex or less safe APIs, less portable code, etc.

It is also going against the flow. Quoting the Microsoft documentation:

> Use UTF-8 character encoding for optimal compatibility between web apps and other *nix-based platforms (Unix, Linux, and variants), minimize localization bugs, and reduce testing overhead.
> UTF-8 is the universal code page for internationalization and is able to encode the entire Unicode character set. It is used pervasively on the web, and is the default for *nix-based platforms.

We should not pretend that lone surrogates in paths are not a reality of life for some users somewhere. However, these users can use native APIs to handle these cases in ways that fit their needs. They are doing it today and can continue to do it tomorrow.

With luck, by restricting the set of conflicting requirements we try to contend with, we can simplify the design of `std::arguments` (P3474R0 [4]), `std::path_view` (P1030R8 [2], P2645R1 [5]), `std::environment` (P1750R1 [3]) and other papers dealing with system APIs.

References

[1] Peter Bindels, Hana Dusikova, and Jeremy Rifkin. P3655R0: zstring_view. `https://wg21.link/p3655r0`, 3 2025.

[2] Niall Douglas. P1030R8: std::filesystem::path_view. `https://wg21.link/p1030r8`, 12 2024.

[3] Klemans Morgenstern, Jeff Garland, Elias Kosunen, and Fatih Bakir. P1750R1: A proposal to add process management to the c++ standard library. `https://wg21.link/p1750r1`, 10 2019.

[4] Jeremy Rifkin. P3474R0: std::arguments. `https://wg21.link/p3474r0`, 10 2024.

[5] Victor Zverovich. P2645R1: path_view: a design that took a wrong turn. `https://wg21.link/p2645r1`, 11 2024.

[N5008] Thomas Köppe *Working Draft, Standard for Programming Language C++* `https://wg21.link/N5008`