P0430R0 – File system library on non-POSIX-like operating systems

Jason Liu, Hubert Tong

2016-09-12

Document Number:	P0430R0
Date:	2016-09-12
Reply-to:	Jason Liu <jasonliu.development@gmail.com>, Hubert Tong <hubert.reinterpretcast@gmail.com>
Authors:	Jason Liu, Hubert Tong
Audience:	LWG/LEWG

1. Introduction

The specification of File system in C++17 is intuitive and easy to understand on a POSIX operating system. Although it gives some leeway for non-POSIX operating systems in 27.10.2.1 [fs.conform.9945], the wording for the file system library is still either confusing or under-specified, or too restrictive/over-specified in some areas for some file systems. This paper is intended to point out some potential file system issues on operating systems with file systems which do not map well to the current model behind the file system library, and propose a way to fix it without affecting the behavior of existing use cases.

2. Technical specification

The following is relative to N4606.

2.1 The term "pathname" in 27.10.8 [class.path] is ambiguous in some context.

It is strongly implied in the standard that the exposition-only member of std::filesystem::path, **pathname**, contains a string representation of the path in native pathname format. For example, **native()** returns "pathname", and **generic_string()** returns "pathname, reformatted according to the generic pathname format".

But in some cases, **pathname** appears to refer to a generic pathname format string without specifying it. For example, **pathname** is used in 27.10.8.4.9 [path.decompose] several times (e.g., in the description of root_name()) in such a way that the interpretation established above, that **pathname** is a native pathname format string, makes no sense.

This is an issue when the generic pathname format and native pathname format are not the same (27.10.8.2 [path.cvt]).

Add the following specification to 27.10.8.2.1 [path.fmt.cvt]:

Specifications for path appends, path concatenation, path modifiers, path decomposition and path query are in terms of the generic pathname format. An implementation needs to make whatever changes necessary to the pathname in native pathname format to produce the specified change in the generic pathname format, or return query result for pathname in terms of the generic pathname format.

2.2 Extra flag in path constructors is needed to distinguish whether source is in native pathname format, or generic pathname format.

This is not an issue for POSIX operating system, since same string have same meaning in both native pathname format and generic pathname format. But for some operating systems, a string could be designed in a way that it's both accepted as native pathname format and generic pathname format, but the interpretation as being in a particular format yields a different abstract path than the interpretation in the other format. Therefore, constructors need an extra argument to understand if the string_type argument is in native pathname format, generic pathname format, or leave it for the implementation to define which format it is in.

Add the following enum class:

enum class pathname_format

Constant	Meaning
native	The format of the string_type argument is in native format.
generic	The format of the string_type argument is in generic format.
auto	Implementation-defined which format of the string_type argument is considered to be in. [*Note*: The implementation may rely on the content of the string_type argument to determine the format. *–end note*]

Modify path constructors in 27.10.8.4.1 [path.construct]:

path(string_type&& source, pathname_format format = pathname_format::auto);

4 Effects: Constructs an object of class path with pathname having the value of source, converting format if required. source is left in a valid but unspecified state.

template <class Source>
path(const Source& source, pathname_format format = pathname_format::auto);
template <class InputIterator>
path(InputIterator first, InputIterator last, pathname_format format = pathname_format::auto);

5 Effects: Constructs an object of class path, storing the effective range of source (27.10.8.3) or the range [first, last] in pathname, converting format and encoding if required (27.10.8.2).

template <class Source>
path(const Source& source, const locale& loc, pathname_format format = pathname_format::auto);
template <class InputIterator>
path(InputIterator first, InputIterator last, const locale& loc, pathname_format format = pathname_format::auto);

6 Requires: The value type of Source and InputIterator is char.

7 Effects: Constructs an object of class path, storing the effective range of source or the range [first, last] in pathname, after converting format if required and after converting the encoding as follows:

(7.1) — If value_type is wchar_t, converts to the native wide encoding (27.10.4.10) using the codecvt<wchar_t, char, mbstate_t> facet of loc.

(7.2) — Otherwise a conversion is performed using the codecvt<wchar_t, char, mbstate_t> facet of loc, and then a second conversion to the current narrow encoding.

2.3 Root-name related issue.

2.3.1 For an operating system that have both a Unix-style file system and another “native” file system, the path to access non-Unix-style files can be very different from a generic pathname format path. In this case, they might want to continue access to Unix-style files as a normal POSIX operating system does, and design a generic pathname format for users to access their native files as well. An implementation for the generic pathname format that accesses native files could choose a multi-character name with a colon as a “root-name”, so that when we see that special root-name, we know that this path in generic pathname format is trying to access a native file, not a Unix file. In this case, root-name is not necessarily a starting location for absolute paths. It is a name used to disambiguate the remainder of the path.

Modify root-name definition in 27.10.8.1 [path.generic]:

root-name:
An operating system dependent name that ~~identifies the starting location for absolute paths~~ can be used to disambiguate the remainder of the path. [Note: A root-name can be used to identify the starting location for absolute paths; it can also be used to invoke alternative pathname resolution. Many operating systems define a name beginning with two directory-separator characters as a root-name that identifies network or other resource locations. Some operating systems define a single letter followed by a colon as a drive specifier – a root-name identifying a specific device such as a disk drive. —end note]

2.3.2 Another issue related to root-name: https://cplusplus.github.io/LWG/lwg-active.html#2664
In issue 2664, it was proposed that if argument path p have a root-name, then operator/= have undefined behavior. But non-POSIX operating system could design its generic pathname for native file type to have a root-name and use it in some creative way. For example, if argument p has a root-name, then p's root-name have to be removed before appending.

Modify specification of path appends in 27.10.8.4.3 [path.append]:
…
path& operator/=(const path& p);
2 Effects: Appends path::preferred_separator to pathname unless:
(2.1) — an added directory-separator would be redundant, or
(2.2) — an added directory-separator would change a relative path into an absolute path [Note: An empty path is relative.—end note] , or
(2.3) — p.empty() is true, or
(2.4) — *p.native().cbegin() is a directory-separator.
Then appends p.native() to pathname. If p.has_root_name(), result is implementation-defined.
3 Returns: *this.
…

Modify specification of path concatenation in 27.10.8.4.4 [path.concat]:
…
If the value type of effective-argument would not be path::value_type , the actual argument or argument range is first converted (27.10.8.2.2) so that effective-argument has value type path::value_type . If

p.has_root_name(), result is implementation-defined.

2 Returns: *this .

…

2.4 Some filesystem operations' behavior are over-specified.

2.4.1 In 27.10.4.1 [fs.def.absolute.path], standard already acknowledges that the elements of a path that determine if it is absolute are operating system dependent. However, in 27.10.15.1 [fs.op.absolute], we are still trying to return an absolute path based on what absolute path should be on a POSIX operating system.

Modify the specification of absolute function in 27.10.15.1 [fs.op.absolute]:

…

Returns: ~~An absolute path (27.10.4.1) composed according to Table 122.~~ If status(p).type() is an implementation-defined file type, then the returned path is implementation-defined. Otherwise, an absolute path (27.10.4.1) composed according to Table 122.

…

2.4.2 If https://cplusplus.github.io/LWG/lwg-active.html#2678 get into the standard, an operating system could have implementation-defined file types. However, some functions are not friendly to those implementation-defined file types. For those functions, an error is required if a file with an implementation-defined file type is encountered; therefore, more useful behavior is precluded.

Modify the following specification of directory_iterator in 27.10.13 [class.directory_iterator]:

An object of type directory_iterator provides an iterator for a sequence of directory_entry elements representing the files in a ~~directory~~ directory-like file type.

Modify the following specification of copy in 27.10.15.3 [fs.op.copy]:

…

Effects are then as follows:

— If status(f).type() or status(t).type() is an implementation-defined file type, then the effects are implementation-defined.

(3.3) — ~~An~~ Otherwise, an error is reported as specified in Error reporting (27.10.7) if:

(3.3.1) — !exists(f), or

(3.3.2) — equivalent(from, to), or

(3.3.3) — is_other(f) || is_other(t), or

(3.3.4) — is_directory(f) && is_regular_file(t).

(3.4) — Otherwise, if is_symlink(f), then:

…

Modify the following specification of file_size in 27.10.15.14 [fs.op.file_size]:

Returns:

— If status(p).type() is an implementation-defined file type, then the returns, including error reporting, are implementation-defined.

— Otherwise, if !exists(p) || !is_regular_file(p) an error is reported (27.10.7).

— Otherwise, the size in bytes of the file p resolves to, determined as if by the value of the POSIX stat structure member st_size obtained as if by POSIX stat(). The signature with argument ec returns static_cast<uintmax_t>(-1) if an error occurs.

Modify the following specification of status in 27.10.15.35 [fs.op.status]:

…

(6.2) — Otherwise,

— If the attributes indicate an implementation-defined file type, let A be the enum name of the implementation-defined file type, returns file_status(file_type::A, prms).

(6.2.1) — If the attributes indicate a regular file, as if by POSIX S_ISREG, returns file_status(file_type::regular, prms). [Note: file_type::regular implies appropriate <fstream> operations would succeed, assuming no hardware, permission, access, or file system race errors. Lack of file_type::regular does not necessarily imply <fstream> operations would fail on a directory. — end note]

…