P0430R1 – File system library on non-POSIX-like operating systems

Jason Liu, Hubert Tong

2016-11-24

Document Number:	P0430R1
Date:	2016-11-24
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 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.

Modify synopsis of Class `path` in 27.10.8 [class.path]:

```
namespace std::filesystem {
class path {
public:
using value_type = see below ;
using string_type = basic_string<value_type>;
static constexpr value_type preferred_separator = see below ;
// pathname format
enum format;
```

...

Add a new section to 27.10.10.x [fs.enum]:

27.10.10.x Enum `path::format`

The `enum` type `path::format` is a bitmask type (17.5.2.1.3 [bitmask.types]) with the elements shown in Table X. When a value of the bitmask type has exactly one element set, the character sequence is interpreted as a pathname in the corresponding format. Otherwise, which of the pathname formats used to interpret the character sequence is determined in an implementation-defined manner. [*Note:* The implementation may inspect the content of the character sequence to determine the format. For POSIX-based systems native and generic formats are equivalent and the character sequence should always be interpreted in the same way. *—end note*]

Table X — Enum `path::format`

enum `path::format`

Name	**Meaning**
`native`	The native pathname format.
`generic`	The generic pathname format.

Modify path constructors in 27.10.8.4.1 [path.construct]:

```
path(string_type&& source, path::format fmt = {});
```

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

```
template <class Source>
```

```
  path(const Source& source, path::format fmt = {});
template <class InputIterator>
  path(InputIterator first, InputIterator last, path::format fmt = {});
```

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

```
template <class Source>
  path(const Source& source, const locale& loc, path::format fmt = {});
template <class InputIterator>
  path(InputIterator first, InputIterator last, const locale& loc,
path::format fmt = {});
```

…

2.2 Root-name is over-specified.

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 an absolute path; 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 pathname resolution. [*Note:* 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 Some filesystem operations' behavior are over-specified.

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] p1:

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

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

…

3 *Effects:* Before the first use of `f` and `t`:

(3.1) — If

`(options & copy_options::create_symlinks) != copy_options::none ||`
`(options & copy_options::skip_symlinks) != copy_options::none`

then `auto f = symlink_status(from)` and if needed `auto t = symlink_status(to)`.

(3.2) — Otherwise, `auto f = status(from)` and if needed `auto t = status(to)`.

Effects are then as follows:

— If `f.type()` or `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 [fs.err.report]) 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 `!exists(p) || !is_regular_file(p)` an error is reported (27.10.7 [fs.err.report]).

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.

— If `!exists(p)` an error is reported (27.10.7 [fs.err.report]).

— Otherwise, if `is_regular_File(p)`, 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.

— Otherwise, the result is implementation-defined.

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

…

6 *Returns:*

(6.1) — If `ec != error_code()`:

(6.1.1) — If the specific error indicates that `p` cannot be resolved because some element of the path does not exist, returns `file_status(file_type::not_found)`.

(6.1.2) — Otherwise, if the specific error indicates that `p` can be resolved but the attributes cannot be determined, returns `file_status(file_type::unknown)`.

(6.1.3) — Otherwise, returns `file_status(file_type::none)`.

[*Note:* These semantics distinguish between `p` being known not to exist, `p` existing but not being able to determine its attributes, and there being an error that prevents even knowing if `p` exists. These distinctions are important to some use cases. *—end note*]

(6.2) — Otherwise,

— If the attributes indicate an implementation-defined file type, returns `file_status(file_type::A, prms)`, where *A* is the constant for the implementation-defined file type.

(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*]

…

3. Acknowledgements

Thanks to Beman Dawes, Jonathan Wakely, Davis Herring, Billy O’Neal and Nicolai Josuttis for assistance on this paper.