

Native handle from file streams

Document #: P1759R2
Date: 2019-12-13
Project: Programming Language C++
Library Evolution
Reply-to: Elias Kosunen
<isocpp@eliaskosunen.com>

Contents

1	Abstract	2
2	Revision History	2
2.1	R2	2
2.2	R1	2
2.3	R0	2
3	Motivation	3
4	Scope	4
5	Design Discussion	4
5.1	Type of <code>native_handle_type</code>	4
5.2	Precondition	5
6	Impact On the Standard and Existing Code	5
7	Implementation	5
8	Prior Art	6
8.1	Discussion	7
8.2	Existing precedent for presence of <code>native_handle</code>	7
9	Technical Specifications	7
9.1	Wording notes	7
9.2	Feature test macro	7
9.3	Wording	8
10	Acknowledgements	11
11	References	11

1 Abstract

This paper proposes adding a new typedef to standard file streams: `native_handle_type`. This type is an alias to whatever type the platform uses for its file descriptors: `int` on POSIX, `HANDLE (void*)` on Windows, and something else on other platforms. This type is a non-owning handle and has generally sane semantics: constexpr default constructability, trivial copyability and standard layout.

Alongside this, this paper proposes adding a concrete member function: `.native_handle()`, returning a `native_handle_type`, to the following class templates:

- `basic_filebuf`
- `basic_ifstream`
- `basic_ofstream`
- `basic_fstream`

2 Revision History

2.1 R2

- Minor touches to wording
 - Refine requirements on `native_handle_type` (remove `equality_comparable`, add constexpr default constructability)
 - Fix some broken references using section numbers in the WD
 - Update reference to the WD
- Editorial fixes

2.2 R1

- Make `native_handle_type` be standard layout
- Add precondition (`is_open() == true`) to `.native_handle()`
- Add feature test macro `__cpp_lib_fstream_native_handle`
- Fix errors with opening the file with POSIX APIs in Motivation (see, we need this paper, fstreams are easier to open correctly!)
- Add additional motivating use case in vectored/scatter-gather IO
- `Regular -> regular`

Incorporate LEWGI feedback from Cologne (July 2019):

- Move to a member function and member typedef
- Make `native_handle` return value not be mandated to be unique
- Add note about how the presence of the members is required, and not implementation-defined (like for thread)

2.3 R0

Initial revision.

3 Motivation

For some operations, using OS/platform-specific file APIs is necessary. If this is the case, one is unable to use iostreams, without reopening the file, with the platform-specific APIs.

For example, if one wanted to query the time a file was last modified on POSIX, one would use `::fstat`, which takes a file descriptor:

```
int fd = ::open("~/foo.txt", O_RDONLY);
::stat s{};
int err = ::fstat(fd, &s);
std::chrono::sys_seconds last_modified = std::chrono::seconds(s.st_mtime.tv_sec);
```

[*Note:* The Filesystem TS introduced the `file_status` structure and `status` function, which returns one. This doesn't solve our problem, because `std::filesystem::status` takes a path, not a native file descriptor (using paths is potentially racy). Also, `std::filesystem::file_status` only contains member functions `type()` and `permissions()`, not one for last time of modification. Extending this structure is out of scope for this proposal, and not feasible for every single possible operation the user may wish to do with OS APIs. — *end note*]

If the user needs to do a single operation not supported by the standard library, they have to make choice between only using OS APIs, or reopening the file every time it's necessary. The former is unfortunate from the perspective of the standard library. The latter is likely to lead to forgetting to close the file, or running into buffering or synchronization issues.

```
// Writing the latest modification date to a file
std::chrono::sys_seconds last_modified(int fd) {
    // See above for POSIX implementation
}

// Today's code

// Option #1:
// Use iostreams by reopening the file
{
    int fd = ::open("~/foo.txt", O_RDONLY); // CreateFile on Windows
    auto lm = last_modified(fd);

    ::close(fd); // CloseFile on Windows
    // Hope the path still points to the file!
    // Need to allocate
    std::ofstream of("~/foo.txt");
    of << std::chrono::format("%c", lm) << '\n';
    // Need to flush
}

// Option #2:
// Abstain from using iostreams altogether
{
    int fd = ::open("~/foo.txt", O_RDWR);
    auto lm = last_modified(fd);

    // Using ::write() is clunky;
    // skipping error handling for brevity
    auto str = std::chrono::format("%c", lm);
    str.push_back('\n');
    ::write(fd, str.data(), str.size());
}
```

```

    // Remember to close!
    // Hope format or push_back doesn't throw
    ::close(fd);
}

// This proposal
// No need to use platform-specific APIs to open the file
{
    std::ofstream of("~/foo.txt");
    auto lm = last_modified(of.native_handle());
    of << std::chrono::format("%c", lm) << '\n';
    // RAII does ownership handling for us
}

```

The utility of getting a file descriptor (or other native file handle) is not limited to getting the last modification date. Other examples include, but are definitely not limited to:

- file locking (`fcntl()` + `F_SETLK` on POSIX, `LockFile` on Windows)
- getting file status flags (`fcntl()` + `F_GETFL` on POSIX, `GetFileInformationByHandle` on Windows)
- vectored/scatter-gather IO (`vread()/vwrite()` on POSIX)
- non-blocking IO (`fcntl()` + `O_NONBLOCK/F_SETSIG` on POSIX)

Basically, this paper would make standard file streams interoperable with operating system interfaces, making iostreams more useful in that regard.

An alternative would be adding a lot of this functionality to `fstream` and `filesystem`. The problem is, that some of this behavior is inherently platform-specific. For example, getting the inode of a file is something that only makes sense on POSIX, so cannot be made part of the `fstream` interface, and is only accessible through the native file descriptor.

Facilities replacing iostreams, although desirable, are not going to be available in the standard in the near future. The author, alongside many others, would thus find this functionality useful.

4 Scope

This paper does *not* propose constructing a file stream or stream buffer from a native file handle. The author is worried of ownership and implementation issues possibly associated with this design.

```

// NOT PROPOSED
#include <fstream>
#include <fcntl.h>

auto fd = ::open(/* ... */);
auto f = std::fstream{fd};

```

This paper also does *not* touch anything related to `FILE*`, namely getting a native handle out of one.

5 Design Discussion

5.1 Type of `native_handle_type`

In this paper, the definition for `native_handle_type` is *much* more strict than in `thread`. For reference, this is the wording from *Native handles* [`thread.req.native`], from [N4835]:

Several classes described in this Clause have members `native_handle_type` and `native_handle`. The presence of these members and their semantics is implementation-defined. [*Note*: These members allow im-

plementations to provide access to implementation details. Their names are specified to facilitate portable compile-time detection. Actual use of these members is inherently non-portable. — *end note*]

During the review of R0 of this paper in Cologne by LEWGI, an implementor mentioned how having the same specification here would make this paper effectively useless. Having the presence of a member be implementation-defined without a feature detect macro was deemed as bad design, which should not be replicated in this paper.

The proposed alternative in this paper, as directed by LEWGI, is allowing a conforming implementation to return an invalid native file handle, if one cannot be retrieved.

5.2 Precondition

The member function `.native_handle()`, as specified in this paper, has a precondition of `.is_open() == true`. The precondition is specified with “Expects”, so breaking it would be UB, and would in practice be enforced with an assert.

An alternative to this would be throwing if the file is not open, or returning some unspecified invalid handle.

6 Impact On the Standard and Existing Code

This proposal is a pure library extension, requiring no changes to the core language. It would cause no existing conforming code to break.

7 Implementation

Implementing this paper should be a relatively trivial task.

Although all implementations surveyed (libstdc++, libc++ and MSVC) use `FILE*` instead of native file descriptors in their `basic_filebuf` implementations, these platforms provide facilities to get a native handle from a `FILE*`; `fileno` on POSIX, and `_fileno + _get_osfhandle` on Windows. The following reference implementations use these.

For libstdc++ on Linux:

```
template <class CharT, class Traits>
class basic_filebuf : public basic_streambuf<CharT, Traits> {
    // ...
    using native_handle_type = int;
    // ...
    native_handle_type native_handle() {
        assert(is_open());
        // _M_file (_basic_file<char>) has a member function for this purpose
        return _M_file.fd();
        // ::fileno(_M_file.file()) could also be used
    }
    // ...
}
```

For libc++ on Linux:

```
template <class CharT, class Traits>
class basic_filebuf : public basic_streambuf<CharT, Traits> {
    // ...
    using native_handle_type = int;
    // ...
    native_handle_type native_handle() {
        assert(is_open());
    }
}
```

```

    // __file_ is a FILE*
    return ::fileno(__file_)
}
// ...
}

```

For MSVC:

```

template <class CharT, class Traits>
class basic_filebuf : public basic_streambuf<CharT, Traits> {
    // ...
    using native_handle_type = HANDLE;
    // ...
    native_handle_type native_handle() {
        assert(is_open());
        // _Myfile is a FILE*
        auto cfile = ::_fileno(_Myfile);
        // _get_osfhandle returns intptr_t, which can be cast to HANDLE (void*)
        return static_cast<HANDLE>(::_get_osfhandle(cfile));
    }
    // ...
}

```

For all of these cases, implementing `.native_handle()` for `ifstream`, `ofstream` and `fstream` is trivial:

```

template <class CharT, class Traits>
class basic_ifstream : public basic_istream<CharT, Traits> {
    // ...
    using native_handle_type =
        typename basic_filebuf<CharT, Traits>::native_handle_type;
    // ...
    native_handle_type native_handle() {
        return rdbuf()->native_handle();
    }
};

// Repeat for ofstream and fstream

```

8 Prior Art

[[Boost.IOStreams](#)] provides `file_descriptor`, `file_descriptor_source`, and `file_descriptor_sink`, which, when used in conjunction with `stream_buffer`, are `std::basic_streambufs` using a file descriptor. These classes can be constructed from a path or a native handle (`int` or `HANDLE`) and can also return it with member function `handle()`.

The Networking TS [[N4734](#)] has members `native_handle_type` and `.native_handle()` in numerous places, including `std::net::socket`. It specifies (in [[socket.reqmts.native](#)]) the presence of these members in a similar fashion to `thread`, as in making their presence implementation-defined. It does, however, recommend POSIX-based systems to use `int` for this purpose.

Niall Douglas's [[P1031](#)] also defined a structure `native_handle_type` with an extensive interface and a member union with an `int` and a `HANDLE`, with a constructor taking either one of these.

8.1 Discussion

There has been some discussion over the years about various things relating to this issue, but as far as the author is aware, no concrete proposal has ever been submitted.

There have been a number of threads on `std-discussion` and `std-proposals`: [\[std-proposals-native-handle\]](#), [\[std-discussion-fd-io\]](#), [\[std-proposals-native-raw-io\]](#), [\[std-proposals-fd-access\]](#). The last one of these lead to a draft paper, that was never submitted: [\[access-file-descriptors\]](#).

The consensus that the author took from these discussions is, that native handle support for iostreams would be very much welcome.

An objection was raised by Billy O’Neal to being able to retrieve a native file handle from a standard file stream:

[This] also would need to mandate that the C++ streams be implemented directly such that there was a 1:1 native handle relationship, which may not be the case. For instance, a valid implementation of C++ iostreams would be on top of `cstdio`, which would not have any kind of native handle to expose.

– Billy O’Neal: [\[std-proposals-billy-oneal\]](#)

Every implementation surveyed did implement `basic_filebuf` on top of C `stdio`, but these platforms also provide functionality for getting a file descriptor out of a `FILE*`. On every platform, file I/O is ultimately implemented on top of native APIs, so not providing access to a file descriptor from a `FILE*` would be rather unfortunate. Should such a platform exist, they probably don’t have a conforming C++ implementation anyway.

Additionally, as directed by LEWGI, `.native_handle()` can just return an invalid handle, if the implementation really can’t get a valid one corresponding to the file.

8.2 Existing precedent for presence of `native_handle`

Types <i>with</i> a standard way of getting the native handle	Types <i>without</i> a standard way of getting the native handle
- <code>std::thread</code>	- <code>std::fstream</code> / <code>std::filebuf</code>
- <code>std::mutex</code>	- <code>FILE*</code>
- Networking TS [N4734] types (e.g. <code>std::net::socket</code>)	
- LLFIO [P1031] types	

This paper would move `std::fstream` and `std::filebuf` from the right column to the left, where it arguably ought to belong.

9 Technical Specifications

9.1 Wording notes

The wording is based on [\[N4835\]](#).

Because of the new section `[file.native]`, some non-stable references using section numbers are broken. This paper fixes some of them, namely the ones near to the parts already changing. It is unclear whether some references not fixed here would be broken should this wording be applied.

9.2 Feature test macro

This paper proposes adding a feature test macro, called `__cpp_lib_fstream_native_handle`.

9.3 Wording

9.3.1 Add the following section into *File-based streams* [file.streams]

This section is to come between [fstream.syn] and [filebuf].

Note to editor: Replace ? with the appropriate section number. As of [N4835], that would be 29.9.2.

???. Native handles [file.native]

- 1 Several classes described in this section have a member `native_handle_type`.
- 2 The type `native_handle_type` serves as a type representing a platform-specific handle to a file. It is trivially copyable and standard layout, models `semiregular`, and has a `constexpr` default constructor.
- 3 [*Note:* For operating systems based on POSIX, `native_handle_type` is `int`. For Windows-based operating systems, `native_handle_type` is `HANDLE`. — *end note*]

9.3.2 Modify *Class template basic_filebuf* [filebuf]

```
namespace std {
    template<class charT, class traits = char_traits<charT>>
    class basic_filebuf : public basic_streambuf<charT, traits> {
    public:
        using char_type    = charT;
        using int_type     = typename traits::int_type;
        using pos_type     = typename traits::pos_type;
        using off_type     = typename traits::off_type;
        using traits_type  = traits;
+   using native_handle_type = implementation-defined; // see [file.native]

        // ...

-   // 29.9.2.3, members
+   // [filebuf.members], members
        bool is_open() const;
        basic_filebuf* open(const char* s, ios_base::openmode mode);
        basic_filebuf* open(const filesystem::path::value_type* s,
                            ios_base::openmode mode); // wide systems only; see 29.9.1
        basic_filebuf* open(const string& s,
                            ios_base::openmode mode);
        basic_filebuf* open(const filesystem::path& s,
                            ios_base::openmode mode);
        basic_filebuf* close();
+   native_handle_type native_handle();

        // ...

+ private:
+   native_handle_type handle; // exposition only
    }
}
```

9.3.3 Modify *Class template basic_filebuf* [filebuf]

- 4 An instance of `basic_filebuf` behaves as described in 29.9.2[filebuf] provided `traits::pos_type` is `fpos<traits::state_type>`. Otherwise the behavior is undefined.

- 5 The underlying file of a `basic_filebuf` has an associated value of type `native_handle_type`, called the *native handle* of the file. Whether the associated *native handle* is unique for each file, is implementation-defined.
- 6 [*Note:* This differs from the native handles of `thread` [`thread.req.native`], the presence of which is implementation-defined. — *end note*]
- 7 In order to support file I/O and multibyte/wide character conversion, conversions are performed using members of a facet, referred to as `a_codecvt` in the following subclauses, obtained as if by

9.3.4 Add to the end of *Member functions* [`filebuf.members`]

```
native_handle_type native_handle();
```

Expects: `is_open()` is true.

Throws: Nothing.

Returns: `handle`.

9.3.5 Modify *Class template basic_ifstream* [`ifstream`]

```
namespace std {
  template<class charT, class traits = char_traits<charT>>
  class basic_ifstream : public basic_istream<charT, traits> {
  public:
    using char_type    = charT;
    using int_type     = typename traits::int_type;
    using pos_type     = typename traits::pos_type;
    using off_type     = typename traits::off_type;
    using traits_type  = traits;
+   using native_handle_type =
+     typename basic_filebuf<charT, traits>::native_handle_type;

    // ...

    // [ifstream.members], members
    basic_filebuf<charT, traits>* rdbuf() const;
+   native_handle_type native_handle();

    // ...
  }
}
```

9.3.6 Add to *Member functions* [`ifstream.members`] after p1

```
native_handle_type native_handle();
```

- 2 *Effects:* Equivalent to: `return rdbuf()->native_handle();`.

9.3.7 Modify *Class template basic_ofstream* [`ofstream`]

```
namespace std {
  template<class charT, class traits = char_traits<charT>>
  class basic_ofstream : public basic_ostream<charT, traits> {
  public:
```

```

using char_type    = charT;
using int_type     = typename traits::int_type;
using pos_type     = typename traits::pos_type;
using off_type     = typename traits::off_type;
using traits_type  = traits;
+ using native_handle_type =
+   typename basic_filebuf<charT, traits>::native_handle_type;

    // ...

    // [ofstream.members], members
    basic_filebuf<charT, traits>* rdbuf() const;
+   native_handle_type native_handle();

    // ...
}
}

```

9.3.8 Add to *Member functions* [ofstream.members] after p1

```
native_handle_type native_handle();
```

² *Effects:* Equivalent to: `return rdbuf()->native_handle();`.

9.3.9 Modify *Class template basic_fstream* [fstream]

```

namespace std {
template<class charT, class traits = char_traits<charT>>
class basic_fstream : public basic_istream<charT, traits> {
public:
    using char_type    = charT;
    using int_type     = typename traits::int_type;
    using pos_type     = typename traits::pos_type;
    using off_type     = typename traits::off_type;
    using traits_type  = traits;
+   using native_handle_type =
+     typename basic_filebuf<charT, traits>::native_handle_type;

    // ...

    // [ofstream.members], members
    basic_filebuf<charT, traits>* rdbuf() const;
+   native_handle_type native_handle();

    // ...
}
}

```

9.3.10 Add to *Member functions* [fstream.members] after p1

```
native_handle_type native_handle();
```

² *Effects:* Equivalent to: `return rdbuf()->native_handle();`.

10 Acknowledgements

Thanks to Niall Douglas for feedback, encouragement and ambitious suggestions for this paper.

Thanks to the rest of the co-authors of [P1750] for the idea after cutting this functionality out, especially to Jeff Garland for providing a heads-up about a possible ABI-break that I totally would've missed, even though it ended up being a non-issue.

Thanks to Michael Park for this his paper markup framework [mpark/wg21].

11 References

- [access-file-descriptors] Bruce S. O. Adams. file streams and access to the file descriptor.
<https://docs.google.com/viewer?a=v&pid=forums&srcid=MTEwODAzNzI2MjM1OTc0MjE3MjkBMDY0OTY1OTUzMjAwNzY0MTA0MjkBakhWMHBFLUNGd0FKATAuMQFpc29jcHAub3JnAXYy&authuser=0>
- [Boost.IOStreams] Jonathan Turkanis. Boost.IOStreams.
https://www.boost.org/doc/libs/1_71_0/libs/iostreams/doc/index.html
- [mpark/wg21] mpark/wg21 on GitHub.
<https://github.com/mpark/wg21>
- [N4734] Jonathan Wakely. 2018. Working Draft, C++ Extensions for Networking.
<https://wg21.link/N4734>
- [N4835] Richard Smith. 2019. Working Draft, Standard for Programming Language C++.
<https://wg21.link/N4835>
- [P1031] Niall Douglas. 2019. Low level file i/o library.
<https://wg21.link/p1031>
- [P1750] Klemens Morgenstern, Jeff Garland, Elias Kosunen, and Fatih Bakir. 2019. A Proposal to Add Process Management to the C++ Standard Library.
<https://wg21.link/p1750>
- [std-discussion-fd-io] File descriptor-backed I/O stream? – std-discussion.
<https://groups.google.com/a/isocpp.org/forum/#!topic/std-discussion/macDvhFDrjU>
- [std-proposals-billy-oneal] Billy O’Neal. Comment on `native_handle` for `basic_filebuf` – std-proposals.
<https://groups.google.com/a/isocpp.org/d/msg/std-proposals/oCEErQbI9sM/rMkAMOkxFvMJ>
- [std-proposals-fd-access] file streams and access to the file descriptor – std-proposals.
<https://groups.google.com/a/isocpp.org/d/topic/std-proposals/XcQ4FZJKDbM/discussion>
- [std-proposals-native-handle] `native_handle` for `basic_filebuf` – std-proposals.
<https://groups.google.com/a/isocpp.org/d/topic/std-proposals/oCEErQbI9sM/discussion>
- [std-proposals-native-raw-io] Native raw IO and `FILE*` wrappers? – std-proposals.
<https://groups.google.com/a/isocpp.org/d/topic/std-proposals/Q4RdFSZggSE/discussion>