

Native handles and standard streams

Document #: P4393R0 [[Latest](#)] [[Status](#)]
Date: 2026-10-14
Project: Programming Language C++
Audience: SG18 (LEWG Incubator)
Reply-to: Jesús Martínez-Cruz
<jmcruz@uma.es>

Contents

1 Abstract	1
2 Changelog	2
3 Introduction	2
4 Is it worthwhile for the standard to offer a solution to this problem?	3
5 The proposal	4
6 Possible implementation	4
7 Open discussion points	6
7.1 Would it not be simpler to use the C standard streams?	6
7.2 Is the overload for <code>basic_istream</code> really necessary?	6
7.3 Is it appropriate to return an invalid value when the stream is not a standard stream?	7
7.4 In which header would the generic function <code>std::native_handle()</code> be declared?	7
8 Conclusions	7
9 Appendix. Repositories consulted	7
10 Note on the use of AI tools	8
11 References	8

1 Abstract

Exposing the native handle that represents an operating system resource, and that is managed inside an object of the C++ standard library, is not new. It is a very useful tool for allowing greater flexibility in our programs, when system calls have to be used that are not yet standardized in C++ or are unlikely ever to be. In C++26, this technique is used once again by adding the public member function `native_handle()` to the classes `basic_filebuf`, `basic_istream`, `basic_ofstream` and `basic_fstream`, which gives access to the native handle of a file [[P1759R6](#)]. In this way, the handle obtained can be used in operating system calls that are familiar to a systems programmer on POSIX (such as `fcntl()`, `read()`...), Windows (`ReadFile()`, `WaitForMultipleObjects()`...), or others. This paper proposes extending this strategy so that the native handle of the files that represent the C++ standard streams (`cin/wcin`, `cout/wcout`, `cerr/wcerr` and `clog/wclog`) can also be obtained, while unifying the way any other native handle is accessed.

Table 1: Obtaining native handles in C++26 (before) and with the proposed `std::native_handle()` (after)

Before	After
<pre>std::ofstream log("log.txt"); auto f = log.native_handle(); // C++26 #ifdef _WIN32 HANDLE out = GetStdHandle(STD_OUTPUT_HANDLE); #else int out = STDOUT_FILENO; #endif</pre>	<pre>std::ofstream log("log.txt"); auto f = std::native_handle(log); auto out = std::native_handle(std::cout);</pre>

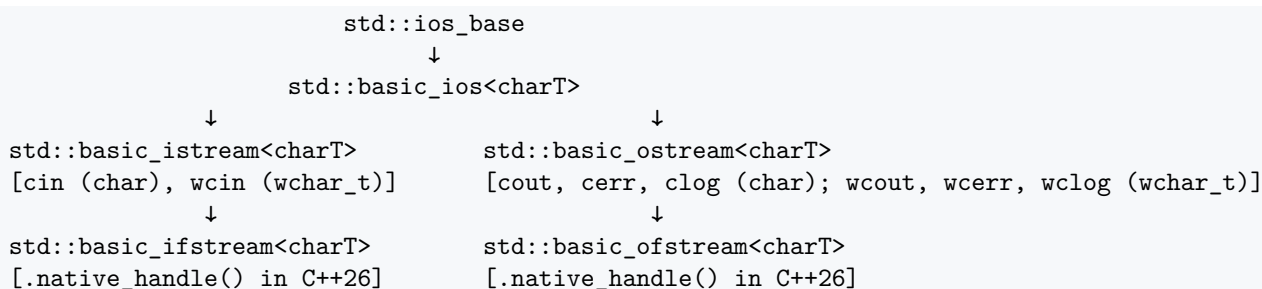
2 Changelog

— R0: initial version of the paper

3 Introduction

Obtaining the native handle associated with an object, which the operating system uses internally, is a very useful tool for interacting with parts of the system’s own API, especially those that would be difficult to standardize as part of the C++ standard library. Recently, in order to expose the native handle associated with files, the inclusion of paper P1759R6 in the standard was approved for C++26. As a result, a new public member function `native_handle()` is added to the classes `basic_filebuf`, `basic_ifstream`, `basic_ofstream` and `basic_fstream`, giving access to the native handle that represents the file descriptor [P1759R6].

However, operating systems also treat the keyboard, the screen and the error console as files identified by their own handle, which allows them to be included in system calls in a unified way. The C++ standard streams (the global objects `cin/wcin`, `cout/wcout`, `cerr/wcerr` and `clog/wclog`) cannot benefit from P1759R6, since their types are `basic_istream` and `basic_ostream`:



Thus, thanks to P1759R6, `native_handle()` is available for file streams, but the eight standard stream objects are left out. As a consequence, in C++26 it will not be possible to write code such as the following in a unified way:

```
/* POSIX: multiplexing two channels (a FIFO and standard input) with poll */
std::ifstream my_fifo("this_is_a_named_pipe_in_unix");
std::array<pollfd, 2> fds = {{
    {my_fifo.native_handle(), POLLIN, 0},
    // ERROR: does not exist; STDIN_FILENO has to be written instead
    {std::cin.native_handle(), POLLIN, 0}
}};
if (poll(fds.data(), fds.size(), -1) > 0) {
    char buf[512];
    for (auto& p : fds)
        if (p.revents & POLLIN) {
```

```

        auto n = read(p.fd, buf, sizeof buf);
        //...
    }
}

/* Windows: finding out what type of object each channel is connected to */
std::ofstream log("log.txt");
DWORD t1 = GetFileType(log.native_handle());
// ERROR: does not exist; GetStdHandle(STD_OUTPUT_HANDLE) has to be written instead
DWORD t2 = GetFileType(std::cout.native_handle());
bool interactive = (t2 == FILE_TYPE_CHAR);

```

4 Is it worthwhile for the standard to offer a solution to this problem?

Currently, the way to obtain the handle of a standard stream depends on the platform. For example, on POSIX one of the constants `STDIN_FILENO`, `STDOUT_FILENO` or `STDERR_FILENO` is used, or else a call to `fileno()` on `stdin`, `stdout` or `stderr`. On Windows, however, a call is made to `GetStdHandle()` with the argument `STD_INPUT_HANDLE`, `STD_OUTPUT_HANDLE` or `STD_ERROR_HANDLE`.

After a search in widely used C++ libraries (see the appendix), we find that this question is usually solved through conditional compilation. For example, this code taken from the Asio library (in the file `include/asio/detail/impl/handler_tracking.ipp`) illustrates it:

```

#elif defined(ASIO_WINDOWS)
    HANDLE stderr_handle = ::GetStdHandle(STD_ERROR_HANDLE);
    DWORD bytes_written = 0;
    ::WriteFile(stderr_handle, line, length, &bytes_written, 0);
#else // defined(ASIO_WINDOWS)
    ::write(STDERR_FILENO, line, length);
#endif // defined(ASIO_WINDOWS)

```

In other libraries, the situation is similar. For example, when Catch2 (in the file `src/catch2/internal/catch_console_colour.cpp`) needs to know whether standard output is a console in order to decide whether to use colors, the Windows part calls:

```

GetConsoleScreenBufferInfo( GetStdHandle( STD_OUTPUT_HANDLE ), &csbiInfo );

```

and the part for the remaining systems calls:

```

useColour = useColour && isatty( STDOUT_FILENO );

```

Even in some other library, such as spdlog (in its file `include/spdlog/sinks/stdout_sinks-inl.h`), the handle for Windows is obtained with:

```

::_get_osfhandle(::_fileno(file_))

```

As can be seen, conditional compilation solves the problem well and is trivial. But if we had a unified way to access the standard streams through the standard library, which is what resolves the specific platform being worked on, the different compilation branches that have to be maintained in each project would be eliminated, and these are moreover not limited to the POSIX and Windows cases. For example, in googletest (file `googletest/include/gtest/internal/gtest-port.h`) its own function `FileNo()`, which returns the descriptor associated with a `FILE*`, has four separate definitions: for Windows Mobile, for the rest of Windows, for ESP8266 and for the remaining systems. Likewise, both `{fmt}` and `spdlog` need special treatment to call `fileno()` on some systems: in `{fmt}` (file `src/os.cc`) its definition as a macro is first removed, with the comment “`fileno` is a macro on OpenBSD.”, and in `spdlog` (file `include/spdlog/details/os-inl.h`) `fileno(f)` is written instead of `::fileno(f)` for OpenBSD and AIX, with the comment “OpenBSD and AIX doesn’t compile with `::` before the `fileno(..)`”.

However, in none of the repositories consulted has any code been found that obtains the handle from the C++ standard streams; instead, all of them start from the platform’s own constant or call, or else from a C

`FILE*`. Doing so from the C++ standard streams is precisely what this proposal puts forward.

5 The proposal

It is clear that the design of the class hierarchy for stream I/O follows clearly differentiated semantics at each level, and that objects of type `basic_{i,o}stream` and derived types have no reason to own and manage an internal operating system resource.

Hence it only makes sense to expose the member function `native_handle()` in the classes proposed in [P1759R6], which have been incorporated into the C++26 standard. However, it would be possible to consider a free function `std::native_handle()`, overloaded with different input arguments, that allows the native handle corresponding to the standard streams to be obtained without modifying their classes. So that the call is written in the same way for a file stream as for a standard stream, this function would also accept any type that has the member `native_handle()`, to which it simply delegates, so that it would return the native handle not only for files, but also for other types of the standard library, such as `thread`, `jthread`, `stacktrace_entry`...

In the overloads for streams, the function will return the handle that the operating system associates with the standard input of the process when it is passed `cin` or `wcin`, that of standard output when it is passed `cout` or `wcout`, and that of standard error when it is passed `cerr`, `wcerr`, `clog` or `wclog`. The stream object serves only to identify which of the three standard channels of the process is requested, so the result does not depend on the state of the stream, not even when its buffer has been replaced with `rdbuf()`. For any other object of type `basic_istream` or `basic_ostream`, the function will return an invalid value.

A possible example of use of this unified interface could be the following:

```
std::ofstream file("test.bin");
auto fd1 = std::native_handle(file);
auto fd2 = std::native_handle(std::cin);
std::jthread t([](){ std::println("thread!"); });
auto tid = std::native_handle(t);
```

The direct inspiration for this proposal is the functions `std::size()`, `std::ssize()` or `std::data()`, which achieve a similar goal [cppreference.data], [cppreference.size].

6 Possible implementation

```
namespace std {

    // exposition only: int on POSIX, HANDLE on Windows [P1759R6]
    using stream-handle = basic_filebuf<char>::native_handle_type;

    // exposition only: value meaning the stream is not one of the standard streams
#ifdef _WIN32
    inline stream-handle invalid-stream-handle() noexcept { return INVALID_HANDLE_VALUE; }
#else
    constexpr stream-handle invalid-stream-handle() noexcept { return -1; }
#endif

    // exposition only: standard handles of the process
#ifdef _WIN32
    inline stream-handle std-in() noexcept { return GetStdHandle(STD_INPUT_HANDLE); }
    inline stream-handle std-out() noexcept { return GetStdHandle(STD_OUTPUT_HANDLE); }
    inline stream-handle std-err() noexcept { return GetStdHandle(STD_ERROR_HANDLE); }
#else
    constexpr stream-handle std-in() noexcept { return STDIN_FILENO; }
    constexpr stream-handle std-out() noexcept { return STDOUT_FILENO; }
    constexpr stream-handle std-err() noexcept { return STDERR_FILENO; }
#endif
}
```

```

// generic function: any type with a member native_handle()
template<class T>
    requires requires(T&& t) { std::forward<T>(t).native_handle(); }
constexpr decltype(auto) native_handle(T&& obj)
    noexcept(noexcept(std::forward<T>(obj).native_handle()))
{
    return std::forward<T>(obj).native_handle();
}

// function for the standard input streams
template<class charT, class traits>
stream-handle native_handle(const basic_istream<charT, traits>& s) noexcept
{
    if constexpr (same_as<basic_istream<charT, traits>, istream>)
        { if (addressof(s) == addressof(cin)) return std-in(); }
    else if constexpr (same_as<basic_istream<charT, traits>, wistream>)
        { if (addressof(s) == addressof(wcin)) return std-in(); }
    return invalid-stream-handle();
}

// function for the standard output streams
template<class charT, class traits>
stream-handle native_handle(const basic_ostream<charT, traits>& s) noexcept
{
    if constexpr (same_as<basic_ostream<charT, traits>, ostream>) {
        if (addressof(s) == addressof(cout)) return std-out();
        if (addressof(s) == addressof(cerr) ||
            addressof(s) == addressof(clog)) return std-err();
    } else if constexpr (same_as<basic_ostream<charT, traits>, wostream>) {
        if (addressof(s) == addressof(wcout)) return std-out();
        if (addressof(s) == addressof(wcerr) ||
            addressof(s) == addressof(wclog)) return std-err();
    }
    return invalid-stream-handle();
}

// this last overload is defined so that the call is not ambiguous (see notes)
template<class charT, class traits>
stream-handle native_handle(const basic_iostream<charT, traits>&) noexcept
{
    return invalid-stream-handle();
}
}

```

Some notes on this implementation:

- The type returned by the overloads for standard streams is the `native_handle_type` of [\[file.native\]](#), which the standard describes as “a platform-specific native handle to a file” (`int` on POSIX, `HANDLE` on Windows), and which is named in the code with the alias `stream-handle`.
- The stream overloads return an invalid value when the object is not one of the eight standard streams, this being `-1` on POSIX and `INVALID_HANDLE_VALUE` on Windows. However, on Windows this decision has a consequence that is set out later, in the open discussion points.
- It can be seen that the signatures of `native_handle()` given in the standard are not uniform: `mutex` and its variants, `condition_variable`, `thread` and `jthread` declare `native_handle_type native_handle()`; although its presence in these classes is “implementation-defined” [\[thread.req.native\]](#); `stacktrace_entry` declares `constexpr native_handle_type native_handle() const noexcept`; and `basic_filebuf`, `basic_ifstream`, `basic_ofstream` and `basic_fstream`

declare `native_handle_type native_handle() const noexcept;`. Our proposed generic function is `constexpr`, which allows it to inherit whatever each member permits, and it is available for a type exactly when its member is. However, the stream overloads have not been proposed as `constexpr` in the reference implementation because on Windows a call to `GetStdHandle()` is required at run time.

- The rules that are intended to be guaranteed, and that the implementation satisfies, are three. (1) If the type has a member `native_handle()`, that member is called, even if the type derives from a stream. In this case, the generic function is an exact match, as opposed to the derived-to-base conversion that the others need. (2) Any other stream, with any `charT` and any `traits`, returns the handle if the object is one of the eight standard streams, and the invalid value otherwise. (3) Ambiguous calls are ruled out: if a type derives from both `basic_istream` and `basic_ostream` (such as `basic_stringstream`), and the third overload does not exist, the following situation arises:

```
std::ostringstream os;
std::native_handle(os); // compiles; returns the invalid value
std::stringstream ss;
std::native_handle(ss); // does not compile: ambiguous call
```

The third overload exists here solely to resolve the ambiguity. Because it refers to a closer base class, it is a better candidate than the other two, and it always returns the invalid value because the eight standard streams are of type `basic_istream` or `basic_ostream` and none is of type `basic_iostream`, so an object of this type can never be one of them. With it, the two cases above behave in the same way and rule (2) is satisfied.

- On the impact on existing code. As happens when any free function is added to namespace `std`, code that defines its own free function `native_handle()` and calls it unqualified with an argument of a standard library type will also find the new function through argument-dependent lookup (ADL), and the new function will be the one chosen when the user's function takes its argument by reference to `const` and is called with a non-`const` object. No case of this pattern has been found either in the repositories of the appendix or in a search of the public index of gregg-app.com.

The reference implementation is available at RefImpl, and has been tested with GCC, Clang and MSVC.

7 Open discussion points

7.1 Would it not be simpler to use the C standard streams?

Since the result of the call to `native_handle()` does not depend on the state of the stream, the objects `std::{cin, cout, cerr, clog}` are only serving us to identify the standard channel. The same could be achieved by accepting the C streams:

```
auto h = std::native_handle(stdin);
```

A single overload would be enough, which would moreover work for any `FILE*` (`fileno()` on POSIX; `_get_osfhandle(_fileno())` on Windows), or a function with no arguments such as `this_process::stdio()`, which was proposed in P1750R1. Both are cheaper to implement, and the first is the form from which the code of existing libraries starts.

The reason for proposing, even so, the C++ standard streams in the overloads is consistency. If a programmer uses `native_handle()` on their `std::{ifstream, ofstream, fstream}` objects, it is also reasonable to think that they will use the objects for the C++ standard streams, not `stdin` or `stdout`. In this way, the friction of having to switch to the C streams or to a platform-dependent constant is removed, and the teaching of the use of the language is improved.

7.2 Is the overload for `basic_iostream` really necessary?

If we remove it, the call with a `basic_stringstream` (or any type derived from `basic_iostream` without a member `native_handle()`) does not compile, with an ambiguous call error. Another option is to declare it as `= delete`. In this way the call would not compile either, but with the error indicating that the function is deleted, which would better express that, for that type, there is never a handle.

However, in both alternatives, a `basic_ostringstream` would still compile and return the invalid value, whereas a `basic_stringstream` would not compile. This is why the proposal argues that all streams that are not standard streams should behave in the same way.

7.3 Is it appropriate to return an invalid value when the stream is not a standard stream?

On POSIX there is no possible confusion, since the standard streams return 0, 1 or 2 and the invalid value is -1. However, on Windows the function `GetStdHandle()` returns `INVALID_HANDLE_VALUE` when it fails [[GetStdHandle](#)], which is the same value that is used to indicate that the stream is not a standard stream, so that whoever calls `std::native_handle()` with a reference to an arbitrary stream and receives that value cannot distinguish whether the object is not a standard stream or whether it is one but the operating system has not been able to provide its handle.

An alternative would be for the stream overloads to return a `std::optional<native_handle_type>`, where `nullopt` would always mean that the object is not a standard stream. However, the returned type would no longer match that of the member `native_handle()`, which is precisely the uniformity that this proposal seeks, and hence the invalid value is kept.

7.4 In which header would the generic function `std::native_handle()` be declared?

Our proposal is that it be declared in `<utility>`, and that it also be available when `<iostream>` or `<fstream>` is included, so that anyone working with streams does not have to include anything else. For the three stream overloads, it would be enough to declare them in `<iostream>`, which is also where the eight standard stream objects they need are declared. Although their return type is that of [[file.native](#)], the implementation can name it directly, without needing to include `<fstream>`.

With regard to the cost of including the generic function in `<utility>` and the three overloads in `<iostream>`, no measurable difference in compilation times was observed (with GCC 14.2, GCC 16.0.1 and Clang 19.1) in a test with eleven calls in one translation unit, a number similar to the maximum number of occurrences per file in the repositories consulted (see the appendix).

8 Conclusions

C++26 allows the native handle of file streams to be obtained, but not that of the standard streams. This proposal seeks to close that gap with a free function `std::native_handle()` that returns the native handle associated with `cin`, `cout`, `cerr`, `clog` and their wide counterparts without modifying their classes, something that other languages already offer, such as Rust (`std::io::stdin().as_raw_fd()`) [[Rust.AsRawFd](#)] or Go (`os.Stdin.Fd()`) [[Go.File.Fd](#)]. So that access is unified, the same function also accepts any object whose class has the member `native_handle()`, such as `thread`, `jthread`, `stacktrace_entry`... This small addition can help to improve the teaching and use of C++ in systems programming.

9 Appendix. Repositories consulted

The following table lists nine widely used C++ libraries in whose code the handle of a standard stream is obtained, both on POSIX and on Windows. The libraries were chosen by hand because they are well known, so they do not constitute a random sample, and each one was consulted at the commit indicated, as its repository stood on 2 October 2026. The search was carried out with the following regular expressions, excluding documentation files (`*.md` and `*.txt`) in the first one:

```
POSIX:      STD(IN|OUT|ERR)_FILENO|fileno\s*\(\s*std(in|out|err)\s*\)
Windows:    GetStdHandle\s*\(
```

The POSIX and Windows columns indicate the number of files in which each form appears, including the tests and the copies of other libraries that some projects carry inside them, and the Both column the number of files in which the two appear at the same time. These figures should be interpreted as a minimum, since the search does not find code that writes the numbers 0, 1 and 2 directly, nor code that calls `fileno()` on a variable.

Library	Commit	POSIX	Windows	Both
fmtlib/fmt	ea0660a	2	1	1
gabime/spdlog	77336f7	0	1	0
libuv/libuv	49b1c06	7	3	1
chriskohlhoff/asio	8806a68	15	1	1
boostorg/process	cfbaf1f	14	2	2
abseil/abseil-cpp	1a8a725	2	0	0
facebook/folly	418a3a8	18	2	0
google/googletest	bc548fd	2	2	1
catchorg/Catch2	222e233	3	2	2

In this search, the file in which the handle of a standard stream appears most often is `test/test-spawn.c` in `libuv`, with thirteen occurrences. No other file in the repositories consulted exceeded nine occurrences.

10 Note on the use of AI tools

In the preparation of this document, Claude (Anthropic) has been used as a support tool to review the consistency of the text, carry out the repository searches collected in the appendix, prepare the tests for the reference implementation, format the document and translate it into English. The design of the proposal, the decisions taken and the final content are the responsibility of the author.

11 References

- [cppreference.data] CppReference. `std::data`.
<https://en.cppreference.com/w/cpp/iterator/data>
- [cppreference.size] CppReference. `std::size`, `std::ssize`.
<https://en.cppreference.com/w/cpp/iterator/size>
- [GetStdHandle] Microsoft. `GetStdHandle` function.
<https://learn.microsoft.com/en-us/windows/console/getstdhandle>
- [Go.File.Fd] Go Language Documentation. `func (*File) Fd`.
<https://pkg.go.dev/os#File.Fd>
- [P1750R1] Klemens Morgenstern, Jeff Garland, Elias Kosunen, Fatih Bakir. 2019-10-07. A Proposal to Add Process Management to the C++ Standard Library.
<https://wg21.link/p1750r1>
- [P1759R6] Elias Kosunen. 2023-05-17. Native handles and file streams.
<https://wg21.link/p1759r6>
- [RefImpl] Jesús Martínez-Cruz. Reference implementation for `std::native_handle`.
<https://godbolt.org/z/ozhKs63cq>
- [Rust.AsRawFd] The Rust Standard Library. `Trait AsRawFd`.
<https://doc.rust-lang.org/std/os/fd/trait.AsRawFd.html>