file streams and access to the file descriptor

Bruce S. O. Adams

Definitions

This document uses the following terms:

- Native File Descriptor - meaning an underlying possibly OS specific handle to file
- FD - a Posix file descriptor (a special case of a native file descriptor)
- HANDLE - a windows file handle (a special case of a native file descriptor)

See also: https://en.wikipedia.org/wiki/File_descriptor

Motivation

- Interoperation between C++ streams and operations using posix File Descriptors
- Interoperation between C++ streams and C stdio functions based on FILE*

Native File Descriptors

Some operations on a POSIX system, notably those making use of fcntl(), require access to the underlying file descriptor.

The C++ standard does not mandate any mechanism to contruct a stream buffer from a file descriptor or to obtain the underlying file descriptor though it is almost certainly accessible in an implementation on a POSIX system.

This motivation also applies to some non Posix based operating systems. For example the Windows API includes functions like LockFile() which make use of a "HANDLE" to a file.

file locking is used here as a driving example. It is not the only use case. The primary use case considered is to provide interoperability with the POSIX functions fcntl(), lockf() & fdopen().

Other use cases could be collated here in support.

Example operations based on fcntl():

- file locking using F_SETLK()
- non-blocking IO (using O_NONBLOCK or F_SETSIG).

Note that non-blocking IO is not a good example in this context as C++ streams currently provide no direct support for non-blocking IO.

Requirements:

- construct a (derived class of) streambuf to implement an fstream based on a native file descriptor
- obtain a file descriptor as used to implement an fstream like object based on native IO functions.

C stdio functions based on FILE*

A seemingly unrelated issue is to allow C streams to interoperate seemlessly with C stdio operations based on FILE*. A library function written in C may use or return a handle to an open device as a FILE*. To use such a function with an C fstream requires IO operations to be synchronised between them and possibly to maintain multiple handles to the same underlying file.

A FILE* mandates C stdio behaviour which may include buffering (see for example https://stackoverflow.com/questions/2423628/whats-the-difference-between-a-file-descriptor-and-file-pointer). It may be that there are use cases where buffering identical to stdio is desired.

Requirements:

- construct a (derived class of) streambuf to implement an fstream based on a FILE*
- obtain a FILE* from a streambuf used to implement a stdio based fstream like object.

Current Solutions

Many implementations of libstdc++ provide their own extensions allowing a buffer to be constructed from a FILE* or native file descriptor directly and like-wise to obtain the FILE* or native file descriptor where it was used to construct them or is otherwise available due to implementation details (for example where std::filebuf is implemented in terms of FILE* or FD).

The canonical and arguably more portable method of doing this is for the user to derive a new streambuf themselves based on stdio or native functions. There are a few variants of this available off the shelf (e.g. stdio_buffer in GNUs libstdc++).

This is the typical answer given to the FAQ of how to do this on stackoverflow and elsewhere.

This situation is non-optimal as:

- non-standard extensions are inherantly non-portable
- classes derived from std::streambuf or std::filebuf providing the same core functionality are not standardised

Better Solution

- Either standardise the non-standard extensions
- Provide standard library classes

Link between requirements for FILE* and native file handles

The requirements show that the FILE* case is very similar to the native file descriptor case.

Note that a solution to one problem is not necessarily a solution to the other.

On Posix it is possible (with limitations) to convert a FD to a FILE* using fdopen() and a FILE* to an FD using fileno(). On Posix platforms therefore it can be argued that a solution to one problem is potentially a solution to the other.

Relation to previous proposals

Though this must come up often. I have not seen a recent concrete proposal. The most recent discussions I can find are:

The one leading to this proposal: https://groups.google.com/a/isocpp.org/forum/#!topic/std-proposals/XcQ4FZJKDbM

from 2014:

https://groups.google.com/a/isocpp.org/forum/?fromgroups#!searchin/std-proposals/posix/std-proposals/Q4RdFSZggSE/RcRWo1S7dKsJ https://groups.google.com/a/isocpp.org/forum/#!topic/std-discussion/macDvhFDrjU

from 2013:

https://groups.google.com/a/isocpp.org/forum/?fromgroups#!searchin/std-proposals/native_handle/std-proposals/oCEErQbI9sM/Oy57TLzCpj4J

Proposal

- A new opaque type representing a native file descriptor
- a stdio_filebuf for synchronising C++ stream IO with C stdio based on FILE*
- a native_filebuf exposing a native file descriptor
- add a new parameter to the std::basic_fstream template declaring the type of filebuf used
- redefine std::fstream as typedef std::basic_fstream<std::filebuf,char,char_traits>
- add stdio_fstream as typedef std::basic_fstream<stdio_filebuf,char,char_traits>
- add native_fstream as typedef std::basic_fstream<native_filebuf,char,char_traits>
- add a new abstract base class "fstream_interface" (better name required) that extends std::basic_iostream with open(), is_open() & close() functions as provided by std::fstream and is implemented by std::basic_fstream

Is is implementation defined whether std::stdio_filebuf, std::native_filebuf and std::filebuf are different types. Some could be typedefs of others provided that the additional interfaces of stdio_filebuf and native_filebuf are provided.

std::native_file_descriptor

It is proposed to add an opaque type representing a native_file_descriptor.

An implementation is free to add its own fields to this. A future version of the C++ standard might consider making this less opaque. For an example of what might be included see include/afio/v2.0/native_handle_type.hpp in the AFIO library proposed for Boost by Niall Douglas - git://github.com/ned14/boost.afio.git

This proposal mandates only the minimum that is required to implement a native_filebuf.

The original suggestion was:

```
struct std::native_file_descriptor
{
#ifdef _POSIX_C_SOURCE
   // return the underlying FD
   int fileno() const;
#endif
};
```

The POSIX definition is required to allow portable access to the FD, for example to use fcntl().

A potentially fatal flaw with this design is that it assumes that a native_file_descriptor on Posix must be implemented in terms of an FD (or something translatable to one). Conceivably there could be Posix compliant platforms on which a native_filebuf is not a posix_filebuf but something lower-level. This might occur, for example, on an OS providing a Posix emulation layer (such as windows). Such a system would have to supply a posix_filebuf to meet the original use case that is distinct from the native_filebuf proposed. The implementation survey done so far has not found one but I believe it must be allowed for. Therefore I propose:

```
#ifdef _POSIX_C_SOURCE
struct std::posix_file_descriptor
{
  int fileno() const;
  // other content implementation defined
};
class posix_filebuf ...
typedef basic_fstream<posix_filebuf,char,charTrais> posix_fstream;
#endif
```

It would then be left implementation defined whether native_filebuf == posix_filebuf

Use of _POSIX_C_SOURCE does introduce a coupling between ISO C++ and POSIX but it is a trivial one. An alternative to consider is introducing a posix namespace and making this std::posix::file_descriptor.

Boost ASIO includes a posix::stream_descriptor (see http://www.boost.org/doc/libs/1_47_0/doc/html/boost_asio/reference/posix__stream_descriptor.html). This does not appear in the networking TS proposal - http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4656.pdf

std::stdio_filebuf

The interface for std::stdio_filebuf shall be identical to the interface for std::filebuf except:

- replacing std::filebuf with std::stdio_filebuf in all functions
- The addition of a constructor accepting a FILE*: stdio_filebuf::stdio_filebuf(FILE* stdioFile);
- The addition of file() function returning a FILE* FILE* stdio_filebuf::file() const;

We could mandate a wrapper to setvbuf(). However, this is left to a future proposal as setvbuf() can be invoked already once the FILE* is exposed.

An issue to be resolve here is whether and how this should be "synchronised".

Consider GNU libstdc++ which provides both stdio_filebuf vs stdio_sync_filebuf We should also consider the analogy with sync_with_stdio() and ios::tie()

Implementation

An Implementation of this proposal should be provided.

To be useful as a reference it should compile under at least clang & gcc on both Linux and Windows and be dual licensed under BSD and LGPL.

Several trials implementations could be produced to test solutions given different starting assumptions.

- std::filebuf is built on FILE* on a Posix system
- std::filebuf is built on FILE* on a non-Posix system
- std::filebuf is built on a posix FD
- std::filebuf is built on a native file descriptor (e.g. a windows HANDLE)

Problems to be addressed before this can be made into a proposal to ISO

- syncronisation of stdio_filebuf with stdio
- how should std::filebuf be implemented?
    - what to do when the implementation of std::filebuf is based on FILE* and thus similar or identical to the proposed stdio_filebuf
    - what to do when the implementation of std::filebuf is based on native_file_descriptor and thus similar or identical to the proposed native_filebuf
    - what to do when the implementation of std::filebuf is similar or identical to both the proposed native_filebuf and stdio_filebuf (e.g. provides constructors from both FILE* and FD)

Implications for the definition of a file stream

The current canonical usage of a custom streambuf representing a native file is something like:

```
void someFunc()
{
  custom_native_filebuf buffer;
  iostream someFile(&buffer);
}
```

This is problematic in that:

- someFile does not own or contain buffer
- someFile does not implement the interface of fstream. Notably the functions open() and close().

An alternative is:

```
void someFunc()
{
  custom_native_filebuf buffer;
  std::fstream someFile;
  someFile.set_rdbuf(buffer);
}
```

This is problematic in that:

- someFile does not own or contain buffer
- someFile constructs a std::filebuf which is redundant.

A function that operates on a file stream should be able to accept a stream that uses either std::filebuf, std::stdio_filebuf or std::native_filebuf. It must be legal for that function to use functions such as open() which are currently available in std::fstream but not in std::iostream.

Do we:

a) extend std::fstream to permit its rdbuf to be any of the three? b) provide std::stdio_fstream() to go with std::stdio_filebuf() c) provide a new base class std::filestream() as a base class of for all three?

stdio_filebuf, native_filebuf and std::filebuf could have different sizes. We would prefer these to be members rather than pointers. So we add the filebuf type as template parameter

```
typedef std::basic_fstream<std::filebuf,char> std::fstream;
typedef std::basic_fstream<native_filebuf,char> native_fstream;
typedef std::basic_fstream<stdio_filebuf,char> stdio_fstream;
```

It should also be possible to have a function that operates on a file based stream and can accept any of these three types. Therefore we propose introducing an abstract base class for a basic_fstream which simply augments basic_iostream with open(), is_open() and close() methods

The name fstream_interface is used here pre bikesheding.

Alternatives to this proposal:

- standardise some existing non-standard extensions
- provide higher-level interfaces for the use cases considered
- do nothing. Leave the status quo as is.

Standardising non-standard extensions

We could require that:

- std::filebuf provides a constructor accepting a FILE*
- std::filebuf provides a function to obtain a FILE*
- std::filebuf provides a constructor accepting a native file descriptor
- std::filebuf provides a function to obtain a native file descriptor

These in effect mandate that std::filebuf == stdio_filebuf and/or std::filebuf == native_filebuf

What constraints do these place on an implementation?

Provide higher level interfaces instead

Providing higher level interfaces is fraught with implementation defined issues which would need to be worked out in full. Such proposals might require something like this proposal as a low-level interface on which to build or they might use an entirely different mechanism e.g. http://www.boost.org/doc/libs/1_44_0/doc/html/boost/interprocess/file_lock.html

Do nothing

I argue against the "do nothing" option as this is a frequent requirement on POSIX based systems as evidenced by questions on stack overflow and elsewhere going back many years. See: https://www.google.co.uk/search?q=fstream+and+file+descriptor

notably: https://stackoverflow.com/questions/2746168/how-to-construct-a-c-fstream-from-a-posix-file-descriptor

Implementing a fdbuf is also described in Nicolai Josuttis's "The C++ Standard Library".

Discussions

How to represent a native file descriptor

Possible options a the native file descriptor representation are:

- a naked file descriptor. i.e. an integer on Posix
- a FILE*
- a struct wrapping a naked file descriptor.

A naked file descriptor on Posix is just an integer and therefore a very leaky abstraction. It should thus be replaced by a less leaky abstraction in a higher level API.

A FILE* mandates stdio buffering which is likely not equivalent to native IO. This is therefore ruled out.

The 'file_descriptor' should therefore be a struct which is opaque to the standard but allows implementation defined access to the low level representation.

In order to solve the original use case, that of being able to call fcntl() on Posix, it is necessary to define how to obtain an FD. Without this each implementation on a POSIX conforming platform would be free to define its own members and definitions for doing so rendering native_file_descriptor non-portable on even Posix systems.

C++ and Posix

Defining a Posix binding for C is out of scope. Some believe this is a job for POSIX and some for the C standard. I believe this is a job for the interested members of the C++ committee because such a binding should be built on top of the Posix C bindings as a pure library extension.

The preference in C++ standardisation is rightly to standardise the functionality rather than the OS binding. For example the filesystem TS standardises access to file-systems and would be wrong to require a Posix binding e.g. opendir() in its interface.

Similarly it can be argued that file locking must be implemented without direct reference to fcntl(), therefore exposing an FD so that fcntl() can be used to implement file locking is potentially misplaced.

The C++ standard should instead aim to standardise an interface for locking files. This might be a standalone interface (i.e. not using iostream) as is provided in AFIO or it could be built upon iostreams for example fstream.lock() or fstream << lock().

While this is true to eliminate the native_file_descriptor use case entirely would require the C++ standard to support everything that is currently done in Posix using fcntl(). It might even need to consider cases covered by the ioctl() system call which predate it.

My view is that OS bindings are orthogonal. They provide a foundation upon which these higher level services may be built. They also continue the C++ tradition of being able to ascend and descend the level of abstraction as appropriate to the programming task required.

Implementation of std::filebuf

I argue that std::filebuf is typically implemented as a native_filebuf or (perhaps more naively) as a stdio_filebuf. If this is the case then std::filebuf could be a typedef for either stdio_filebuf or native_filebuf In the case of Posix given fileno() and fdopen() is is also possible that the same class is all three of std::stdio_filebuf std::nativefilebuf and std::filebuf.

I have thus left it implementation defined whether std::stdio_filebuf, std::native_filebuf and std::filebuf are different types. Some could be typedefs of others provided that the additional interfaces of stdio_filebuf and native_filebuf are provided.

Some have argued that they MUST be different types. Presumably this is either so that std::filebuf is a less leaky abstraction and/or so that the standard mandates the complete interface for each class and does not permit implementation defined extensions.

I believe more arguments would be required to justify adding this restriction.

History

For some reason I felt compelled to answer an old question on stack overflow. https://stackoverflow.com/questions/8667581/flock-ing-a-c-ifstream-on-linux-gcc-4-6/45525028#45525028

Because I have seen this discussed several times I had assumed there was a proposal already. This resulted in this discussion when I found none.

https://groups.google.com/a/isocpp.org/forum/#!topic/std-proposals/XcQ4FZJKDbM

The initial proposal considered stdio_filebuf as out of scope.

The hope was that std::filebuf could be extended such that std::filebuf == native_filebuf

However, the assumption that std::filebuf must be implemented in terms of an FD on a Posix compliant sysem need not be true. A conforming implementation could be built upon a C stdio FILE* or upon some other low-level descriptor. Posix might be supported through an emulation layer only.

These considerations inevitably lead to a more complex proposal.

survey of existing filebuf implementations

Only a few open source implementations have been surveyed so far. The visual C++ interface was surveyed using the free tier installation as a canonical non Posix exemplar. Input from those experienced with other systems would be appreciated.

Implementations surveyed:

- apache stdcxx (based on import of roguewave standard library implementation 2005)

- GNU libstdc++3
- clang libcxx-4.0.1
- open watcom v2

To summarise findings. std::filebuf is frequently implemented in terms of FILE*.

This suggests we could make std::filebuf == stdio_filebuf as the default and provide native_filebuf as an optional optimisation.

apache stdcxx

std::filebuf is based on FILE*

constructors are provided for both FILE* and FD:

```
// extension - return the associated file descriptor
int fd () const {
    return _RW::__rw_fileno (_C_file, this->_C_state);
}
```

```
// extension - associate *this with an open file descriptor
basic_filebuf* attach (int __fd) {
    return _C_open (__fd, 0, this->_C_buffer, this->_C_bufsize);
}
```

```
// extension - flush and detach from file descriptor without closing it
int detach () {
    const int __fd = fd ();
    return close (false) ? __fd : -1;
}
```

GNU libstdc++3

Based on gcc 7.1.0.

std::filebuf is based on FILE*

provides extensions

- libstdc++-v3/include/ext/stdio_filebuf.h & - FILE* and FD
- libstdc++-v3/include/ext/stdio_sync_filebuf.h & FILE* only

clang libcxx

std::filebuf is based on FILE*

No extensions provided.

open watcom v2

Included as it is open source and therefore accessible. It may not be in widespread use.

filebuf is based on an FD.

- filebuf can be constructed from an FD.
- an fd() function can be used to obtain the FD.
- an attach() function can be used to associate a filebuf with a new FD.

visual c++

basic_filebuf is built on top of FILE*

MSDN describes:

```
filedesc fd() const;
filebuf* attach( filedesc fd );
```

https://msdn.microsoft.com/en-us/library/aa243812(v=vs.60).aspx https://msdn.microsoft.com/en-us/library/aa243810(v=vs.60).aspx

Though these did not appear the headers I examined.