1 Introduction

At present, some stream output operations guarantee that they will not produce race conditions, but do not guarantee that the effect will be sensible. Some form of external synchronization is required. Unfortunately, without a standard mechanism for synchronizing, independently developed software will be unable to synchronize.

N3535 C++ Stream Mutexes proposed a standard mechanism for finding and sharing a mutex on streams. At the Spring 2013 standards meeting, the Concurrency Study Group requested a change away from a full mutex definition to a definition that also enabled buffering.

N3678 C++ Stream Guards proposed a standard mechanism for batching operations on a stream. That batching may be implemented as mutexes, as buffering, or some combination of both. It was the response to the Concurrency Study Group. A draft of that paper was reviewed in the Library Working Group, who found too many open issues on what was reasonably exposed to the 'buffering' part.

N3665 Uninterleaved String Output Streaming proposed making streaming of strings of length less than BUFSIZ appear uninterleaved in the output. It was a "minimal" functionality change to the existing standard to address the problem. The full Committee chose not to adopt that minimal solution.

N3750 C++ Ostream Buffers proposed an explicit buffering, at the direction of the general consensus in the July 2013 meeting of the Concurrency Study Group. In November 2014 a further updated version N4187 was discussed in Library Evolution in Urbana and it was consensus to work with a library expert to get the naming and wording better suited to LWG expectations. Nico Josuttis volunteered to help the original authors. More information on the discussion is available at LEWG wiki and the corresponding LEWG bug tracker entry (20). This paper address issues raised with N4187. This paper has a change of title to reflect a change in class name, but contains the same fundamental approach.
1.1 Solution

We propose a `basic_osyncstream`, that buffers output operations for a wrapped stream. The `basic_osyncstream` will atomically transfer the contents of an internal stream buffer to a `basic_ostream`'s stream buffer on destruction of the `basic_osyncstream`.

The transfer on destruction simplifies the code and ensures at least some output in the presence of an exception.

The intent is that the `basic_osyncstream` is an automatic-duration variable with a relatively small scope which constructs the text to appear uninterleaved. For example,

```cpp
{  
  osyncstream bout(cout);  
  bout << "Hello, " << "World!" << '\n';  
}
```

or with a single output statement:

```cpp
osyncstream{cout} << "The answer is " << 6*7 << endl;
```

2 Design

Here we list a set of previous objections to our proposal in the form of questions. We then give our reasons why other potential solutions do not work as well as our proposed solution.

**Why can I not just use cout? It should be thread-safe.**

You will not get data races for cout, but that is not true for most other streams. In addition there is no guarantee that output from different threads appears in any sensible order. Coherent output order is the main reason for this proposal.

**Why must osyncstream be an ostream? Could a simple proxy wrapper work?**

To support all existing and user-defined output operators, osyncstream must be an ostream subclass.

**Can you make a flush of the osyncstream mean transfer the characters and flush the underlying stream?**

No, because the point of the osyncstream is to collect text into an atomic unit. Flushes are often emitted in calls where the body is not visible, and hence unintentionally break up the text. Furthermore, there may be more than one flush within the lifetime of an osyncstream, which would impose a performance loss when an atomic unit of text needs only one flush. The design decision is only to flush the underlying stream if the osyncstream was flushed, and only once per atomic transfer of the character sequence. `p0052r5 introduced manipulators to allow both ways, but the default remains not to flush immediately.`

**Can flush just transfer the characters and not flush the underlying stream?**

The flush intends an effect on visible to the user. So, we should preserve at least one flush. The flush may not be visible to the code around the osyncstream, and so its programmer cannot do a manual flush, because attempting to flush the underlying stream that is shared among threads will introduce a data race.
Why do you specify basic_syncbuf? LWG and LEWG thought you wouldn’t need it.
Users use the streambuf interface. Without access to the basic_syncbuf they would not be able call emit() on the underlying streambuf responsible the synchronization. Every stream in the standard is defined as a pair of streambuf/stream, for good reason. Ask Pablo Halpern if you need more to be convinced.

Where will the required lock/mutex be put? Will it be in every streambuf object changing the ABI?
That is one of the reasons why this must be put into the standard library. It is possible to implement with a global map from streambuf* to mutexes as the example code does, however, existing standard library implementations might have already a space for the mutexes (not breaking their ABI), because cout/cerr seem to require one and those might be the most important ones to wrap.

The design follows from the principles of the iostream library. If discussed a person knowledgable about iostream’s implementation is favorable, because of its many legacy design decisions, that would no longer be taken by modern C++ class designers.

As with all existing stream classes, using a stream object or a streambuf object from multiple threads can result in a data race, this is also true for osyncstream. Its use enables sharing the wrapped streambuf object/output stream across several threads, if all concurrently using threads apply a local osyncstream around it.

We follow typical stream conventions of basic_ prefixes and typedefs.
The constructor for osyncstream takes a non-const reference to a basic_ostream obtaining its stream buffer or a basic_streambuf* or can be put around such a stream buffer directly. Calling the emit() member function or destroying the osyncstream may write to the stream buffer obtained at construction.

The wording below permits implementation of basic_osyncstream with either a stream_mutex from N3535 or with implementations suitable for N3665, e.g. with Posix file locks [PSL]

2.1 Items to be discussed by LEWG/LWG
— Naming of the manipulators
— Should the manipulators be in header <osyncstream> instead of globally available?
— Should the manipulator-hook member function of basic_syncbuf be public, or should the manipulators be made friends instead. However, without the accessor users can not directly force the stream buffer to pass through every sync.
— re-check wording.
— Should the Allocator be part of the stream type, or should it simply use polymorphic_allocator<CharT>, consistent with P0339? Pablo thinks the latter would be easier to use.
— What should be the delivery vehicle for this feature: C++20 or the concurrency TS?
3 Wording

This wording is relative to the current C++ working draft. It could be integrated into a Concurrency TS accordingly.

Changes will happen in section 30 mainly.

In section 20.5.1.1 Library contents [contents] add an entry to table 16 (cpp.library.headers) for the new header <syncstream>. Because of the new header, there is no need for a feature test macro.

3.1 30.1 General [input.output.general]

Insert a new entry in table 106 Input/output library summary (tab:iostreams.lib.summary)

| 30.x  | Synchronized output stream  | <syncstream> |

3.2 30.3.1 Header <iosfwd> synopsis [iosfwd.syn]

Insert the following declarations to the synopsis of <iosfwd> in the namespace std.

```cpp
template <class charT,
          class traits = char_traits<charT>,
          class Allocator = allocator<charT>>
class basic_syncbuf;
using syncbuf = basic_syncbuf<char>;
using wsyncbuf = basic_syncbuf<wchar_t>;

template <class charT,
          class traits = char_traits<charT>,
          class Allocator = allocator<charT>>
class basic_osyncstream;
typedef basic_osyncstream<char> osyncstream;
typedef basic_osyncstream<wchar_t> wosyncstream;
```

3.3 30.7.5.4 Standard basic_ostream manipulators [ostream.manip]

Add the following two manipulators.

```cpp
template <class charT, class traits>
basic_ostream<charT, traits>& emit_on_flush(basic_ostream<charT, traits>& os);

1 Effects: If os is a basic_osyncstream<charT, traits, Allocator> reference, calls dynamic_cast<basic_syncbuf<charT, traits, Allocator>>*(os.rdbuf())->set_emit_on_sync(true). Otherwise this manipulator has no effect. [Note: To work around the issue that the Allocator template argument can not be deduced, implementations can introduce an intermediate base class to basic_syncbuf that takes care its emit_on_sync flag. —end note]

2 Returns: os.

template <class charT, class traits>
basic_ostream<charT, traits>& noemit_on_flush(basic_ostream<charT, traits>& os);
```
Effects: If os is a basic_osyncstream<charT, traits, Allocator> reference, calls dynamic_cast<basic_osyncbuf<charT, traits, Allocator>*>(os.rdbuf())->set_emit_on_sync(false). Otherwise this manipulator has no effect.

Returns: os.

```
template <class charT, class traits>
  basic_ostream<charT, traits>& flush_emit(basic_ostream<charT, traits>& os);
```

Effects: flush(os), if os is a basic_osyncstream<charT, traits, Allocator> reference, dynamic_cast<basic_osyncbuf<charT, traits, Allocator>*>(os.rdbuf())->emit(). Returns: os.

### 3.4 30.x Synchronized output stream [syncstream]

*Insert this new section 30.x in chapter 30 [input.output]*

#### 3.4.1 30.x.1 Header <synstream> synopsis [syncstream.syn]

```cpp
namespace std {
  template <class charT,
    class traits = char_traits<charT>,
    class Allocator = allocator<charT>>
    class basic_syncbuf;
  using syncbuf = basic_syncbuf<char>;
  using wsyncbuf = basic_syncbuf<wchar_t>;

  template <class charT,
    class traits,
    class Allocator>
    class basic_osyncstream;
  typedef basic_osyncstream<char> osyncstream;
  typedef basic_osyncstream<wchar_t> wosyncstream;
}
```

The header <syncstream> provides a mechanism to synchronize execution agents writing to the same stream. It defines a class template basic_osyncstream to buffer output and transfer the buffered content into an object of type basic_streambuf<charT,traits> atomically with respect to such transfers by other basic_osyncstream<charT,traits,Allocator> objects referring to the same basic_streambuf<charT,traits> object. The transfer occurs when emit() is called and when the basic_osyncstream<charT,traits,Allocator> object is destroyed.

#### 3.4.2 30.x.2 Class template basic_syncbuf [syncstream.syncbuf]

```cpp
template <class charT,
  class traits = char_traits<charT>,
  class Allocator = allocator<charT>>
  class basic_syncbuf
    : public std::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 allocator_type = Allocator;

using streambuf_type = basic_streambuf<charT,traits>);

explicit basic_syncbuf(streambuf_type* obuf = nullptr)
: basic_syncbuf(obuf,Allocator{});

basic_syncbuf(streambuf_type* obuf,
    Allocator const &allocator);
~basic_syncbuf();

basic_syncbuf& operator=(basic_syncbuf&& rhs);

void swap(basic_syncbuf &other);

bool emit();
streambuf_type* get_wrapped() const noexcept;

allocator_type get_allocator() const noexcept;
void set_emit_on_sync(bool b) noexcept;

protected:
int sync() override;

private:
streambuf_type *wrapped;    // exposition only
bool emit_on_sync{};    // exposition only

};

3.5 30.x.2.1 basic_syncbuf constructors [syncstream.syncbuf.cons]

explicit basic_syncbuf(
    streambuf_type* obuf = nullptr,
    Allocator const &allocator = Allocator{});

Effects: Constructs the basic_syncbuf object and sets wrapped to obuf which will be the final destination of characters inserted into the stream.

Remarks: If obuf == nullptr, output operations on *this will fail. A copy of allocator is used to allocate memory for internal buffers.

Throws: Nothing unless constructing a mutex or allocating memory throws.

[ Note: Subsequent calls to emit() might result in member functions being called on the user-provided stream buffer while a lock is held. — end note ]

Postconditions: get_wrapped() == obuf & get_allocator() == allocator.
basic_syncbuf(basic_syncbuf&& other);

Effects: If other.get_wrapped() == nullptr, equivalent to basic_syncbuf(nullptr, other.get_allocator()) otherwise the state necessary to perform other.emit() is moved to *this.

Postconditions: The value returned by this->get_wrapped() is the value returned by other.get_wrapped() prior to calling this constructor. Output stored in other prior to calling this constructor will be stored in *this afterwards. other.rdbuf()->pbase() == other.rdbuf()->pptr() and other.get_wrapped() == nullptr.

Remarks: This constructor disassociates other from its wrapped stream buffer ensuring destruction of other produces no output.

3.6 30.x.2.2 basic_syncbuf destructor [syncstream.syncbuf.dtor]

~basic_syncbuf();

Effects: Calls this->emit().

Throws: Nothing. If an exception is thrown while writing to the wrapped buffer, that exception is caught and ignored.

3.6.1 30.x.2.3 basic_syncbuf assign and swap [syncstream.syncbuf.assign]

basic_syncbuf& operator=(basic_syncbuf&& rhs) noexcept;

Effects: Calls this->emit(). If rhs.get_wrapped() != nullptr, the state necessary to perform rhs.emit() is moved to *this.

Returns: *this.

Postconditions: rhs.get_wrapped() == nullptr. If allocator_traits<Allocator>::propagate_on_container_move_assignment::value == true, then this->get_allocator() == rhs.get_allocator(); otherwise the allocator is unchanged.

Remarks: This assignment operator disassociates rhs from its wrapped stream buffer ensuring destruction of other produces no output.

void swap(basic_syncbuf& other) noexcept;

Requires: this->get_allocator() == other.get_allocator().

Effects: Exchanges the state of *this and other.

template <class charT, class traits, class Allocator>
void swap(basic_syncbuf<charT,traits,Allocator>& a, 
          basic_syncbuf<charT,traits,Allocator>& b) noexcept;

Effects: As if by a.swap(b).

3.6.2 30.x.2.4 basic_syncbuf member functions [syncstream.syncbuf.mfun]

bool emit();

Effects: Atomically transfers the contents of the internal buffer to the stream buffer *wrapped, so that they appear in the output stream as a contiguous sequence of characters. If and only
if a call was made to this->sync(), wrapped->pubsync() is called.

Returns: true if wrapped != nullptr, all of the buffered characters were successfully transferred, and the call to wrapped->pubsync() (if any) succeeded; otherwise false.

Synchronization: All emit() calls transferring characters to the same stream buffer object appear to execute in a total order consistent with happens-before where each emit() call synchronizes-with subsequent emit() calls in that total order.

Remarks: May call member functions of wrapped while holding a lock uniquely associated with wrapped.

streambuf_type* get_wrapped() const noexcept;

Returns: wrapped.

allocator_type get_allocator() const noexcept;

Returns: A copy of the allocator set in the constructor or from assignment.

void set_emit_on_sync(bool b) noexcept;

Effects: emit_on_sync=b.

3.6.3 30.x.2.5 basic_syncbuf overridden virtual member functions
[syncstream.syncbuf.virtauls]

int sync() override;

Effects: Record the fact that user desires that the wrapped buffer be flushed. If emit_on_sync == true, calls emit(), otherwise the actual flush is delayed until a call to emit().

Returns: if emit() was called and returned false, returns -1 otherwise 0.

3.7 30.x.3 Class template basic_osyncstream [syncstream.osyncstream]

template <class charT,
               class traits,
               class Allocator>
class basic_osyncstream
 : 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 allocator_type = Allocator;
    using streambuf_type = basic_streambuf<charT,traits>;
    using syncbuf_type = basic_syncbuf<charT,traits, Allocator>;

    basic_osyncstream(streambuf_type *obuf, Allocator const &allocator);
    explicit basic_osyncstream(streambuf_type *obuf)
    :basic_osyncstream{obuf, Allocator{}}{}
basic_osyncstream(basic_ostream<charT,traits> &os, Allocator const &allocator)
    :basic_osyncstream(os.rdbuf(),allocator){}
explicit basic_osyncstream(basic_ostream<charT,traits> &os)
    :basic_osyncstream(os,Allocator{}){}
basic_osyncstream(basic_osyncstream&&) noexcept;
~basic_osyncstream();
basic_osyncstream& operator=(basic_osyncstream&&) noexcept;
void emit();
streambuf_type* get_wrapped() const noexcept;
syncbuf_type* rdbuf() const noexcept override { return &sb ; }
private:
    syncbuf_type sb; // exposition only
};

1 Allocator shall meet the allocator requirements [allocator.requirements].

2 [ Example: Use a named variable within a block statement for streaming.

```cpp
    {  
        osyncstream bout(cout);
        bout << "Hello, ";
        bout << "World!";
        bout << endl; // flush is noted
        bout << "and more!\n";
    } // characters are transferred and cout is flushed
    — end example |
```

3 [ Example: Use a temporary object for streaming within a single statement. cout is not flushed.

```cpp
    osyncstream(cout) << "Hello, " << "World!" << '\n';
    — end example |
```

3.7.1 30.x.3.1 basic_osyncstream constructors [syncstream.osyncstream.cons]

basic_osyncstream(
    streambuf_type *buf,
    Allocator const &allocator);

1 Effects: Initializes sb from buf and allocator and initializes the base class with basic_ostream(&sb).

2 [ Note: If wrapped stream buffer pointer refers to a user provided stream buffer then its implementation must be aware that its member functions might be called from emit() while a lock is held. — end note ]

3 Postconditions: get_wrapped() == buf.

basic_osyncstream(basic_osyncstream&& other) noexcept;

4 Effects: Move constructs from other. This is accomplished by move constructing the base class, and the contained basic_syncbuf sb. Next basic_ostream<charT, traits>::set_rdbuf(&sb) is called to install the basic_syncbuf sb.

5 Postconditions: The value returned by this->get_wrapped() is the value returned by os.get_-

wrapped() prior to calling this constructor. nullptr == os.get_wrapped().

### 3.7.2 30.x.3.2 basic_osyncstream destructor [syncstream.osyncstream.dtor]

```cpp
~basic_osyncstream();
```

**Effects:** Calls `emit()` and swallows any exceptions arising.

### 3.7.3 30.x.3.3 Assignment [syncstream.osyncstream.assign]

```cpp
basic_osyncstream& operator=(basic_osyncstream&& other) noexcept;
```

**Effects:** First, calls `this->emit()` and swallows any exceptions arising. Move assigns `sb` from `other.sb`. *[Note: This disassociates `other` from its wrapped stream buffer ensuring destruction of other produces no output. —end note]*

**Postconditions:** Primarily, `nullptr == other.get_wrapped()`. Also, `get_wrapped()` returns the value previously returned by `other.get_wrapped()`.

### 3.7.4 30.x.3.4 Member functions [syncstream.osyncstream.members]

```cpp
void emit();
```

**Effects:** Calls `sb.emit()`. If this call returns `false`, calls `setstate(ios::badbit)`.

**Example:** A flush on a `basic_osyncstream` does not flush immediately:

```cpp
{
    osyncstream bout(cout);
    bout << "Hello," << \n; // no flush
    bout.emit(); // characters transferred; cout not flushed
    bout << "World!" << endl; // flush noted; bout not flushed
    bout.emit(); // characters transferred; cout flushed
    bout << "Greetings." << \n; // no flush
} // characters transferred; cout not flushed
```

**Example:** The function `emit()` can be used to catch exceptions from operations on the underlying stream.

```cpp
{
    osyncstream bout(cout);
    bout << "Hello, " << "World!" << \n;
    try {
        bout.emit();
    } catch ( ... ) {
        // stuff
    }
}
```

**Returns:** `sb.get_wrapped()`.

streambuf_type* get_wrapped() const noexcept;

**Returns:** `sb.get_wrapped()`.
Example: Obtaining the wrapped stream buffer with `get_wrapped()` allows wrapping it again with an `osyncstream`. For example,

```cpp
{  
osyncstream bout1(cout);  
bout1 << "Hello, ";  
  
    osyncstream(bout1.get_wrapped()) << "Goodbye, " << "Planet!" << '\n';  
}
bout1 << "World!" << '\n';
```

produces the uninterleaved output

```
Goodbye, Planet!
Hello, World!
```

— end example |

### 3.8 Implementation

An example implementation is available on https://github.com/PeterSommerlad/SC22WG21_Papers/tree/master/workspace/p0053_basic_osyncstreambuf

### 4 Revisions

Each section lists the revisions in the following version from the version given in the heading.

#### 4.1 D0053R4

This version was only published on the Kona Wiki.

— Translate text to LaTeX

— Added manipulator support so that logging frameworks that get an osyncstream passed can rely on flushes happening.

— Make sure that temporary stream objects can be used safely (it is already in the standard, Pablo!)

— Ensured section/table numbers match current working draft (as of 06/2017 before the mailing)

#### 4.2 P0053R3

— Takes input from Pablo Halpern and re-instantiate the stream buffer that performs the synchronization.

— Split the constructors with a defaulted allocator parameter to one single-argument one being explicit and one non-explicit taking 2 arguments.

#### 4.3 P0053R2

— Remove the "may construct a mutex" notes.

— Remove "may destroy mutex" notes.
— Clarify osyncstream flush behavior in an example.
— Make minor editorial fixes.

4.4 P0053R1

— Provide a typedef for the wrapped stream buffer and use it to shorten the specification as suggested by Daniel Krügler.
— Provide move construction and move assignment and specify the moved-from state to be detached from the wrapped stream buffer.
— Rename get() to rdbuf_wrapped() and provide noexcept specification.
— Changed to explicitly rely on wrapping a stream buffer, instead of an ostream object and adjust explanations accordingly.

4.5 P0053R0

— Add remark to note that exchanging the stream buffer while the stream is wrapped causes undefined behavior and added a note to warn stream buffer implementers about the lock being held in emit(). Call setstate(badbit) if IO errors occur in emit().
— Replace code references to basic_streambuf by the term stream buffer introduced in [stream.buffers].
— The lock is to be associate to the underlying basic_streambuf instead of the basic_stream.
— Added an Allocator constructor parameter.
— Moves destructor example to emit().
— Clarifies wording about synchronization and flushing (several times).
— List the new header in corresponding table.
— Provide type aliases in <iosfwd>.
— Removed copy constructor in favor of providing get().
— Notify that move construction and assignment is deleted.
— Moved class noteflush_streambuf into an implementation note.
— Add a design subsection that states that a header test is a sufficient feature test.

4.6 N4187

— Updated introduction with recent history.
— Rename ostream_buffer to osyncstream to reflect its appearance is more like a stream than like a buffer.
— Add an example of using osyncstream as a temporary object.
— Add an example of a osyncstream constructed with another osyncstream.
— Clarify the behavior of nested osyncstream executions.
— Clarify the behavior of exceptions occurring with the osyncstream destructor.
— Clarify the deferral of flush from the osyncstream's streambuf to the final basic_ostream.
— Limit the number of references to noteflush_stringbuf in anticipation of the committee removing it from the specification.
— Rename noteflush_stringbuf to noteflush_streambuf to hide possible implementation details.
— Change the base class of noteflush_streambuf from basic_stringbuf to basic_streambuf.

4.7 N4069
— Added note to sync as suggested by BSI via email.

4.8 N3978
— Added a Design section.
— Clarify the reference capturing behavior of the ostream_buffer constructors.
— Added noexcept and const as appropriate to members.
— Added note on throwing wrapped streams.
— Change the noteflush_stringbuf public member variable needsflush to a public member query function flushed.
— Removed the public member function noteflush_stringbuf::clear.
— Minor synopsis formatting changes.
— Incorporated feedback from SG1 and Dietmar Kühl in specific in Rapperswil.

4.9 N3892
— Flush the ostream if and only if the ostream_buffer was flushed.
— Add the clear_for_reuse function.
— Change the design from inheriting from basic_ostream to using a noteflush_stringbuf, which is a slightly modified basic_stringbuf. The modification is to note the flush rather than act upon it.

4.10 N3750
— Change name to basic_ostream_buffer and add the usual typedefs.
— Change interface to inherit from basic_ostringstream rather than provide access to a member of that type.
— Add a Revisions section.