p0448r1 - A strstream replacement using span<charT> as buffer

Peter Sommerlad

2017-03-03

Document Number:	p0448r1 (N2065 done right?)
Date:	2017-03-03
Project:	Programming Language C++
Audience:	LWG/LEWG

1 History

Streams have been the oldest part of the C++ standard library and especially strstreams that can use pre-allocated buffers have been deprecated for a long time now, waiting for a replacement. p0407 and p0408 provide the efficient access to the underlying buffer for stringstreams that strstream provided solving half of the problem that strstreams provide a solution for. The other half is using a fixed size pre-allocated buffer, e.g., allocated on the stack, that is used as the stream buffers internal storage.

A combination of external-fixed and internal-growing buffer allocation that strstreambuf provides is IMHO a doomed approach and very hard to use right.

There had been a proposal for the pre-allocated external memory buffer streams in N2065 but that went nowhere. Today, with `span<T>` we actually have a library type representing such buffers views we can use for specifying (and implementing) such streams. They can be used in areas where dynamic (re-)allocation of stringstreams is not acceptable but the burden of caring for a pre-existing buffer during the lifetime of the stream is manageable.

2 Introduction

This paper proposes a class template `basic_spanbuf` and the corresponding stream class templates to enable the use of streams on externally provided memory buffers. No ownership or re-allocation support is given. For those features we have string-based streams.

3 Acknowledgements

— Thanks to those ISO C++ meeting members attending the Oulu meeting encouring me to write this proposal. I believe Neil and Pablo have been among them, but can't remember who

else.

— Thanks go to Jonathan Wakely who pointed the problem of `strstream` out to me and to Neil Macintosh to provide the span library type specification.

— Thanks to Felix Morgner for proofreading.

— Thanks to Kona LEWG small group discussion suggesting some clarifications and Thomas Köppe for allowing me to use using type aliases instead of typedef.

4 Motivation

To finally get rid of the deprecated `strstream` in the C++ standard we need a replacement. p0407/p0408 provide one for one half of the needs for `strstream`. This paper provides one for the second half: fixed sized buffers.

[*Example:* reading input from a fixed pre-arranged character buffer:

```
char input[] = "10 20 30";
ispanstream is{span<char>{input}};
int i;
is >> i;
ASSERT_EQUAL(10,i);
is >> i ;
ASSERT_EQUAL(20,i);
is >> i;
ASSERT_EQUAL(30,i);
is >>i;
ASSERT(!is);
```

— *end example*] [*Example:* writing to a fixed pre-arranged character buffer:

```
char  output[30]{}; // zero-initialize array
ospanstream os{span<char>{output}};
os << 10 << 20 << 30 ;
auto const sp = os.span();
ASSERT_EQUAL(6,sp.size());
ASSERT_EQUAL("102030",std::string(sp.data(),sp.size()));
ASSERT_EQUAL(static_cast<void*>(output),sp.data()); // no copying of underlying data!
ASSERT_EQUAL("102030",output); // initialization guaranteed NUL termination
```

— *end example*]

5 Impact on the Standard

This is an extension to the standard library to enable deletion of the deprecated `strstream` classes by providing `basic_spanbuf`, `basic_spanstream`, `basic_ispanstream`, and `basic_ospanstream` class templates that take an object of type `span<charT>` which provides an external buffer to be used by the stream.

It also proposes to remove the deprecated strstreams [depr.str.strstreams] assuming p0407 is also included in the standard.

6 Design Decisions

6.1 General Principles

6.2 Open Issues (to be) Discussed by LEWG / LWG

— Should arbitrary types as template arguments to `span` be allowed to provide the underlying buffer by using the `byte` sequence representation `span` provides. (I do not think so and some people in LEWG inofficially agree with it). You can always get a span of characters from the underlying byte sequence, so there is no need to put that functionality into spanbuf, it would break orthogonality.

— Should the `basic_spanbuf` be copy-able? It doesn't own any resources, so copying like with handles or `span` might be fine. Other concrete streambuf classes in the standard that own their buffer (basic_stringbuf, basic_filebuf) naturally prohibit copying, where the base class basic_streambuf provides a protected copy-ctor. This version of the paper provides copyability for basic_spanbuf, because the implementation is `=default`. Note, none of the stream classes in the standard is copyable as are the stream classes provided here.

7 Technical Specifications

Remove section [depr.str.strstreams] from appendix D.

Insert a new section 30.x in chapter 30 [input.output] after section 30.8 [string.streams]

7.1 30.x Span-based Streams [span.streams]

This section introduces a stream interface for user-provided fixed-size buffers.

7.1.1 30.x.1 Overview [span.streams.overview]

The header `<spanstream>` defines four class templates and eight types that associate stream buffers with objects of class `span` as described in [span].

Header `<spanstream>` synopsis

```
namespace std {
namespace experimental {
  template <class charT, class traits = char_traits<charT> >
    class basic_spanbuf;
  using spanbuf = basic_spanbuf<char>;
  using wspanbuf = basic_spanbuf<wchar_t>;
  template <class charT, class traits = char_traits<charT> >
    class basic_ispanstream;
  using ispanstream = basic_ispanstream<char>;
  using wispanstream = basic_ispanstream<wchar_t>;
  template <class charT, class traits = char_traits<charT> >
```

```
    class basic_ospanstream;
  using ospanstream = basic_ospanstream<char>;
  using wospanstream = basic_ospanstream<wchar_t>;
  template <class charT, class traits = char_traits<charT> >
    class basic_spanstream;
  using spanstream = basic_spanstream<char>;
  using wspanstream = basic_spanstream<wchar_t>;
}}
```

7.2 30.x.2 Class template basic_spanbuf [spanbuf]

```
namespace std {
  template <class charT, class traits = char_traits<charT> >
  class basic_spanbuf
    : 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;

    // ??, constructors:
    template <ptrdiff_t Extent>
    explicit basic_spanbuf(
      span<charT, Extent> span,
      ios_base::openmode which = ios_base::in | ios_base::out);
    basic_spanbuf(const basic_spanbuf& rhs) noexcept;
    basic_spanbuf(basic_spanbuf&& rhs) noexcept;

    // ??, assign and swap:
    basic_spanbuf& operator=(const basic_spanbuf& rhs) noexcept;
    basic_spanbuf& operator=(basic_spanbuf&& rhs) noexcept;
    void swap(basic_spanbuf& rhs) noexcept;

    // ??, get and set:
    span<charT> span() const noexcept;
    void span(span<charT> s) noexcept;

  protected:
    // ??, overridden virtual functions:
    int_type underflow() override;
    int_type pbackfail(int_type c = traits::eof()) override;
    int_type overflow (int_type c = traits::eof()) override;
    basic_streambuf<charT, traits>* setbuf(charT*, streamsize) override;

    pos_type seekoff(off_type off, ios_base::seekdir way,
                     ios_base::openmode which
                      = ios_base::in | ios_base::out) override;
    pos_type seekpos(pos_type sp,
                     ios_base::openmode which
                      = ios_base::in | ios_base::out) override;
```

```
  private:
    ios_base::openmode mode;  // exposition only
  };

  template <class charT, class traits>
    void swap(basic_spanbuf<charT, traits>& x,
              basic_spanbuf<charT, traits>& y) noexcept;
}
```

1 The class `basic_spanbuf` is derived from `basic_streambuf` to associate possibly the input sequence and possibly the output sequence with a sequence of arbitrary *characters*. The sequence is provided by an object of class `span<charT>`.

2 For the sake of exposition, the maintained data is presented here as:

(2.1) — `ios_base::openmode mode`, has `in` set if the input sequence can be read, and `out` set if the output sequence can be written.

7.3 30.x.2.1 `basic_spanbuf` constructors [spanbuf.cons]

```
template <ptrdiff_t Extent>
explicit basic_spanbuf(
  basic_span<charT, Extent> s,
  ios_base::openmode which = ios_base::in | ios_base::out);
```

1 *Effects:* Constructs an object of class `basic_spanbuf`, initializing the base class with `basic_streambuf()` (??), and initializing `mode` with `which`. Initializes the internal pointers as if calling `span(s)`.

```
basic_spanbuf(basic_spanbuf const & rhs) noexcept;
basic_spanbuf(basic_spanbuf&& rhs) noexcept;
```

2 *Effects:* Copy/Move constructs from the rvalue `rhs`. Both `basic_spanbuf` objects share the same underlying `span`. The sequence pointers in `*this` (`eback()`, `gptr()`, `egptr()`, `pbase()`, `pptr()`, `epptr()`) obtain the values which `rhs` had. The openmode, locale and any other state of `rhs` is also copied.

3 *Postconditions:* Let `rhs_p` refer to the state of `rhs` just prior to this construction.

(3.1) — `span() == rhs_p.span()`

(3.2) — `eback() == rhs_p.eback()`

(3.3) — `gptr() == rhs_p.gptr()`

(3.4) — `egptr() == rhs_p.egptr()`

(3.5) — `pbase() == rhs_p.pbase()`

(3.6) — `pptr() == rhs_p.pptr()`

(3.7) — `epptr() == rhs_p.epptr()`

7.3.1 30.x.2.2 Assign and swap [spanbuf.assign]

```
basic_spanbuf& operator=(basic_spanbuf const & rhs) noexcept;
basic_spanbuf& operator=(basic_spanbuf&& rhs) noexcept;
```

1 *Effects:* After the copy/move assignment `*this` has the observable state it would have had if it had been copy/move constructed from `rhs` (see **??**).

2 *Returns:* `*this`.

```
void swap(basic_spanbuf& rhs) noexcept;
```

3 *Effects:* Exchanges the state of `*this` and `rhs`.

```
template <class charT, class traits, class Allocator>
  void swap(basic_spanbuf<charT, traits>& x,
            basic_spanbuf<charT, traits>& y) noexcept;
```

4 *Effects:* As if by `x.swap(y)`.

7.3.2 30.x.2.3 Member functions [spanbuf.members]

```
span<charT> span() const;
```

1 *Returns:* A `span` object representing the `basic_spanbuf` underlying character sequence. If the `basic_spanbuf` was created only in output mode, the resultant `span` represents the character sequence in the range `[pbase(), pptr())`, otherwise in the range `[eback(), egptr())`. [*Note:* In constrast to `basic_stringbuf` the underlying sequence can never grow and will not be owned. An owning copy can be obtained by converting the result to `basic_string<charT>`. — *end note*]

```
template<ptrdiff_t Extent>
void span(span<charT,Extent> s);
```

2 *Effects:* Initializes the `basic_spanbuf` underlying character sequence with `s` and initializes the input and output sequences according to `mode`.

3 *Postconditions:* If `mode & ios_base::out` is true, `pbase()` points to the first underlying character and `epptr() >= pbase() + s.size()` holds; in addition, if `mode & ios_base::ate` is true, `pptr() == pbase() + s.size()` holds, otherwise `pptr() == pbase()` is true. If `mode & ios_base::in` is true, `eback()` points to the first underlying character, and both `gptr() == eback()` and `egptr() == eback() + s.size()` hold.

[*Note:* Using append mode does not make sense for `span`-based streams. — *end note*]

7.3.3 30.x.2.4 Overridden virtual functions [spanbuf.virtuals]

1 [*Note:* Since the underlying buffer is of fixed size, neither `overflow`, `underflow` or `pbackfail` can provide useful behavior. — *end note*]

```
int_type underflow() override;
```

2 *Returns:* `traits::eof()`.

```
int_type pbackfail(int_type c = traits::eof()) override;
```

3 *Returns:* `traits::eof()`.

```
int_type overflow(int_type c = traits::eof()) override;
```

4 *Returns:* `traits::eof()`.

```
pos_type seekoff(off_type off, ios_base::seekdir way,
                 ios_base::openmode which
                   = ios_base::in | ios_base::out) override;
```

5 *Effects:* Alters the stream position within one of the controlled sequences, if possible, as indicated in Table **??**.

6 For a sequence to be positioned, if its next pointer (either `gptr()` or `pptr()`) is a null pointer and the new offset `newoff` is nonzero, the positioning operation fails. Otherwise, the function determines `newoff` as indicated in Table **??**.

7 If `(newoff + off) < 0`, or if `newoff + off` refers to an uninitialized character outside the span (as defined in **??** paragraph 1), the positioning operation fails. Otherwise, the function assigns `xbeg + newoff + off` to the next pointer `xnext`.

8 *Returns:* `pos_type(newoff)`, constructed from the resultant offset `newoff` (of type `off_type`), that stores the resultant stream position, if possible. If the positioning operation fails, or if the constructed object cannot represent the resultant stream position, the return value is `pos_type(off_type(-1))`.

```
pos_type seekpos(pos_type sp,
                 ios_base::openmode which
                   = ios_base::in | ios_base::out) override;
```

9 *Effects:* Equivalent to `seekoff(off_type(sp), ios_base::beg, which)`.

10 *Returns:* `sp` to indicate success, or `pos_type(off_type(-1))` to indicate failure.

```
basic_streambuf<charT, traits>* setbuf(charT* s, streamsize n);
```

11 *Effects:* If `s` and `n` denote a non-empty span `this->span(span<charT>(s,n));`

12 *Returns:* `this`.

7.4 30.x.3 Class template `basic_ispanstream` [ispanstream]

```
namespace std {
  template <class charT, class traits = char_traits<charT>>
  class basic_ispanstream
    : 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;

    // ??, constructors:
    template <ptrdiff_t Extent>
    explicit basic_ispanstream(
      span<charT, Extent> span,
```

```
    ios_base::openmode which = ios_base::in);
  basic_ispanstream(const basic_ispanstream& rhs) = delete;
  basic_ispanstream(basic_ispanstream&& rhs) noexcept;

  // ??, assign and swap:
  basic_ispanstream& operator=(const basic_ispanstream& rhs) = delete;
  basic_ispanstream& operator=(basic_ispanstream&& rhs) noexcept;
  void swap(basic_ispanstream& rhs) noexcept;

  // ??, members:
  basic_spanbuf<charT, traits>* rdbuf() const noexcept;

  span<charT> span() const noexcept;
      template<ptrdiff_t Extent>
  void span(span<charT> s) noexcept;
private:
  basic_spanbuf<charT, traits> sb; // exposition only
};

template <class charT, class traits>
  void swap(basic_ispanstream<charT, traits>& x,
            basic_ispanstream<charT, traits>& y) noexcept;
}
```

1 The class `basic_ispanstream<charT, traits>` supports reading objects of class `span<charT, traits>`. It uses a `basic_spanbuf<charT, traits>` object to control the associated span. For the sake of exposition, the maintained data is presented here as:

(1.1) — `sb`, the `spanbuf` object.

7.4.1 30.x.3.1 `basic_ispanstream` constructors [ispanstream.cons]

```
template <ptrdiff_t Extent>
explicit basic_ispanstream(
  span<charT, Extent> span,
  ios_base::openmode which = ios_base::in);
```

1 *Effects:* Constructs an object of class `basic_ispanstream<charT, traits>`, initializing the base class with `basic_istream(&sb)` and initializing `sb` with `basic_spanbuf<charT, traits>span, which | ios_base::in)` (??).

```
basic_ispanstream(basic_ispanstream&& rhs);
```

2 *Effects:* Move constructs from the rvalue `rhs`. This is accomplished by move constructing the base class, and the contained `basic_spanbuf`. Next `basic_istream<charT, traits>::set_rdbuf(&sb)` is called to install the contained `basic_spanbuf`.

7.4.2 30.x.3.2 Assign and swap [ispanstream.assign]

```
basic_ispanstream& operator=(basic_ispanstream&& rhs);
```

1 *Effects:* Move assigns the base and members of `*this` from the base and corresponding members of `rhs`.

2 *Returns:* `*this`.

```
void swap(basic_ispanstream& rhs);
```

3 *Effects:* Exchanges the state of `*this` and `rhs` by calling `basic_istream<charT, traits>::swap(rhs)` and `sb.swap(rhs.sb)`.

```
template <class charT, class traits, class Allocator>
  void swap(basic_ispanstream<charT, traits, Allocator>& x,
            basic_ispanstream<charT, traits, Allocator>& y);
```

4 *Effects:* As if by `x.swap(y)`.

7.4.3 30.x.3.3 Member functions [ispanstream.members]

```
basic_spanbuf<charT>* rdbuf() const noexcept;
```

1 *Returns:* `const_cast<basic_spanbuf<charT>*>(&sb)`.

```
span<charT> span() const noexcept;
```

2 *Returns:* `rdbuf()->span()`.

```
template<ptrdiff_t Extent>
void span(span<charT, Extent> s) noexcept;
```

3 *Effects:* Calls `rdbuf()->span(s)`.

7.5 30.x.4 Class template `basic_ospanstream` [ospanstream]

```
namespace std {
  template <class charT, class traits = char_traits<charT>>
  class basic_ospanstream
    : 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;

    // ??, constructors:
    template <ptrdiff_t Extent>
    explicit basic_ospanstream(
      span<charT, Extent> span,
      ios_base::openmode which = ios_base::out);
    basic_ospanstream(const basic_ospanstream& rhs) = delete;
    basic_ospanstream(basic_ospanstream&& rhs) noexcept;

    // ??, assign and swap:
    basic_ospanstream& operator=(const basic_ospanstream& rhs) = delete;
    basic_ospanstream& operator=(basic_ospanstream&& rhs) noexcept;
    void swap(basic_ospanstream& rhs) noexcept;

    // ??, members:
```

```
    basic_spanbuf<charT, traits>* rdbuf() const noexcept;

    span<charT> span() const noexcept;
        template<ptrdiff_t Extent>
    void span(span<charT> s) noexcept;
  private:
    basic_spanbuf<charT, traits> sb; // exposition only
  };

  template <class charT, class traits>
    void swap(basic_ospanstream<charT, traits>& x,
              basic_ospanstream<charT, traits>& y) noexcept;
}
```

1 The class `basic_ospanstream<charT, traits>` supports writing to objects of class `span<charT, traits>`. It uses a `basic_spanbuf<charT, traits>` object to control the associated span. For the sake of exposition, the maintained data is presented here as:

(1.1) — `sb`, the `spanbuf` object.

7.5.1 30.x.4.1 `basic_ospanstream` constructors [ospanstream.cons]

```
template <ptrdiff_t Extent>
explicit basic_ospanstream(
  span<charT, Extent> span,
  ios_base::openmode which = ios_base::out);
```

1 *Effects:* Constructs an object of class `basic_ospanstream<charT, traits>`, initializing the base class with `basic_ostream(&sb)` and initializing `sb` with `basic_spanbuf<charT, traits>span, which | ios_base::out)` (??).

```
basic_ospanstream(basic_ospanstream&& rhs) noexcept;
```

2 *Effects:* Move constructs from the rvalue `rhs`. This is accomplished by move constructing the base class, and the contained `basic_spanbuf`. Next `basic_ostream<charT, traits>::set_rdbuf(&sb)` is called to install the contained `basic_spanbuf`.

7.5.2 30.x.4.2 Assign and swap [ospanstream.assign]

```
basic_ospanstream& operator=(basic_ospanstream&& rhs) noexcept;
```

1 *Effects:* Move assigns the base and members of `*this` from the base and corresponding members of `rhs`.

2 *Returns:* `*this`.

```
void swap(basic_ospanstream& rhs) noexcept;
```

3 *Effects:* Exchanges the state of `*this` and `rhs` by calling `basic_ostream<charT, traits>::swap(rhs)` and `sb.swap(rhs.sb)`.

```
template <class charT, class traits, class Allocator>
  void swap(basic_ospanstream<charT, traits, Allocator>& x,
            basic_ospanstream<charT, traits, Allocator>& y) noexcept;
```

4 *Effects:* As if by `x.swap(y)`.

7.5.3 30.x.4.3 Member functions [ospanstream.members]

```
basic_spanbuf<charT>* rdbuf() const noexcept;
```

1 *Returns:* `const_cast<basic_spanbuf<charT>*>(&sb)`.

```
span<charT> span() const noexcept;
```

2 *Returns:* `rdbuf()->span()`.

```
template<ptrdiff_t Extent>
void span(span<charT, Extent> s) noexcept;
```

3 *Effects:* Calls `rdbuf()->span(s)`.

7.6 30.x.5 Class template `basic_spanstream` [spanstream]

```
namespace std {
  template <class charT, class traits = char_traits<charT>>
  class basic_spanstream
    : public basic_iostream<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;

    // ??, constructors:
    template <ptrdiff_t Extent>
    explicit basic_spanstream(
      span<charT, Extent> span,
      ios_base::openmode which = ios_base::out);
    basic_spanstream(const basic_spanstream& rhs) = delete;
    basic_spanstream(basic_spanstream&& rhs) noexcept;

    // ??, assign and swap:
    basic_spanstream& operator=(const basic_spanstream& rhs) = delete;
    basic_spanstream& operator=(basic_spanstream&& rhs) noexcept;
    void swap(basic_spanstream& rhs) noexcept;

    // ??, members:
    basic_spanbuf<charT, traits>* rdbuf() const noexcept;

    span<charT> span() const noexcept;
        template<ptrdiff_t Extent>
    void span(span<charT> s) noexcept;
  private:
    basic_spanbuf<charT, traits> sb; // exposition only
  };
```

```
  template <class charT, class traits>
    void swap(basic_spanstream<charT, traits>& x,
              basic_spanstream<charT, traits>& y) noexcept;
}
```

1 The class `basic_spanstream<charT, traits>` supports reading from and writing to objects of class `span<charT, traits>`. It uses a `basic_spanbuf<charT, traits>` object to control the associated span. For the sake of exposition, the maintained data is presented here as:

(1.1) — `sb`, the `spanbuf` object.

7.6.1 30.x.5.1 `basic_spanstream` constructors [spanstream.cons]

```
template <ptrdiff_t Extent>
explicit basic_spanstream(
  span<charT, Extent> span,
  ios_base::openmode which = ios_base::out | ios_bas::in);
```

1 *Effects:* Constructs an object of class `basic_spanstream<charT, traits>`, initializing the base class with `basic_iostream(&sb)` and initializing `sb` with `basic_spanbuf<charT, traits>span, which)` (??).

```
basic_spanstream(basic_spanstream&& rhs) noexcept;
```

2 *Effects:* Move constructs from the rvalue `rhs`. This is accomplished by move constructing the base class, and the contained `basic_spanbuf`. Next `basic_istream<charT, traits>::set_rdbuf(&sb)` is called to install the contained `basic_spanbuf`.

7.6.2 30.x.5.2 Assign and swap [spanstream.assign]

```
basic_spanstream& operator=(basic_spanstream&& rhs) noexcept;
```

1 *Effects:* Move assigns the base and members of `*this` from the base and corresponding members of `rhs`.

2 *Returns:* `*this`.

```
void swap(basic_spanstream& rhs) noexcept;
```

3 *Effects:* Exchanges the state of `*this` and `rhs` by calling `basic_iostream<charT, traits>::swap(rhs)` and `sb.swap(rhs.sb)`.

```
template <class charT, class traits, class Allocator>
  void swap(basic_spanstream<charT, traits, Allocator>& x,
            basic_spanstream<charT, traits, Allocator>& y) noexcept;
```

4 *Effects:* As if by `x.swap(y)`.

7.6.3 30.x.5.3 Member functions [spanstream.members]

```
basic_spanbuf<charT>* rdbuf() const noexcept;
```

1 *Returns:* `const_cast<basic_spanbuf<charT>*>(&sb)`.

```
span<charT> span() const noexcept;
```

2 *Returns:* `rdbuf()->span()`.

```
template<ptrdiff_t Extent>
void span(span<charT, Extent> s) noexcept;
```

3 *Effects:* Calls `rdbuf()->span(s)`.

8 Appendix: Example Implementations

An example implementation will become available under the author's github account at: https://github.com/PeterSommerlad/SC22WG21_Papers/tree/master/workspace/Test_basic_spanbuf