Doc. no.: N2065=06-0135
Date: 2006-09-05
Project: Programming Language C++
Reply to: Alberto Ganesh Barbati
<abarbati@iaanus.com>

A proposal to add stream objects based on fixed memory buffers

1 Motivation

The C++03 standard deprecates the `strstream` class templates, while the `stringstream` counterparts are usually considered a replacement. The string-based streams indeed have a better interface, however they have the following drawbacks:

— if the user has initial data stored in a character buffer, using a `basic_istringstream` object to parse the data requires copying the entire buffer into a `basic_string` object

— if the user requires output data to be stored in a previously allocated character buffer (for example a member of some POD-struct used to call a legacy C function), the data must be copied out of the `basic_string` object returned by `basic_stringbuf::str()` and into the buffer.

Consider also that:

— `basic_string` objects might require dynamic allocations on the heap

— manipulation of `basic_string` objects might perform unnecessary hidden copies of the buffer data. For example, as `basic_stringbuf::str()` returns a string by-value, a typical non-refcounted implementation of `basic_string` requires an additional copy of the buffer data

— in the fixed-buffer output scenario, the fact that both `stringstream`s and `strstream`s provide support for growable buffers is both unnecessary and a nuisance[1]. And you are probably going to pay for it!

For these reasons, in the pre-allocated fixed buffer scenario, users may prefer either the deprecated `strstream`s or avoiding streams entirely.

This proposal is about providing a new set of class templates (one buffer and three streams) that specifically address direct reading from and writing to fixed memory buffers. The buffer shall be provided by the user and will always remain under his complete responsibility. In particular, the user shall ensure that the buffer exists for the entire lifetime of the stream buffer object that manages access to it. The proposed templates never try to allocate, grow, shrink, etc. the given buffer. Any read/write operation is done directly from/to the buffer, with no intermediate copy. The proposed templates also don't try to provide any view of the underlying buffer as a string[2]. These assumptions allow to keep both the interface very terse and the implementation very simple and performant.

Although it is conceivable that users can write such templates on their own, there are a few pitfalls in the implementation (for example in `seekoff()` and `seekpos()`) that make them suitable for standardization.

[1]For example, with the proposed `omemstream` class, output operations fail as soon as the buffer space is exhausted. This fact, which could be exploited for example in conjunction with the `exceptions()` member function, can't be obtained with string-based (growable) streams.

[2]In particular, null characters in the buffer don't get any special treatment and there is no active effort to append null-terminators.

2 Impact on the standard

This proposal is a pure extension. All additions are limited to a single new header file <memstream>, which defines the new templates. It does not require changes in the core language and has been implemented in standard C++ (see annex A).

This proposal does not depend on any other library extensions.

3 Proposed text

3.1 Changes in current standard

3.1.1 Changes to clause 27.5.2.4.2 [lib.streambuf.virt.buffer]

In paragraphs 1, 3 and 5 (functions `setbuf()`, `seekoff()`, `seekpos()`), add to the list of forward references (27.7.1.3, 27.8.1.4) a reference to the new clause [lib.membuf.virtuals].

3.2 Additions to standard

The following text should be added to clause 27.

3.2.1 Memory-based streams [lib.memory.streams]

The header <memstream> defines four class templates and six types, that associate stream buffers with static memory buffers.

3.2.1.1 Header <memstream> synopsis

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

  template <class charT, class traits = char_traits<charT> >
    class basic_imemstream;

  template <class charT, class traits = char_traits<charT> >
    class basic_omemstream;

  template <class charT, class traits = char_traits<charT> >
    class basic_memstream;

  typedef basic_membuf<char>         membuf;
  typedef basic_imemstream<char>     imemstream;
  typedef basic_omemstream<char>     omemstream;
  typedef basic_memstream<char>      memstream;

  typedef basic_membuf<wchar_t>      wmembuf;
  typedef basic_imemstream<wchar_t> wimemstream;
  typedef basic_omemstream<wchar_t> womemstream;
  typedef basic_memstream<wchar_t>  wmemstream;
}
```

3.2.1.2 Class template basic_membuf [lib.membuf]

```
namespace std {
  template <class charT, class traits = char_traits<charT> >
  class basic_membuf : public basic_streambuf<charT,traits> {
  public:
    typedef charT char_type;
    typedef typename traits::int_type int_type;
    typedef typename traits::pos_type pos_type;
    typedef typename traits::off_type off_type;
    typedef traits traits_type;

    // Constructors:
    basic_membuf(charT* s, streamsize n,
      ios_base::openmode which = ios_base::in | ios_base::out);

    basic_membuf(const charT* s, streamsize n,
      ios_base::openmode which = ios_base::in);

    // Capacity:
    static streamsize max_size() const;

  protected:
    // Overridden virtual functions:
    virtual pos_type seekoff(off_type off, ios_base::seekdir way,
      ios_base::openmode which = ios_base::in | ios_base::out);

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

    virtual basic_streambuf<charT,traits>* setbuf(charT*, streamsize);

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

1 The class `basic_membuf` is derived from `basic_streambuf` to associate possibly the input sequence and possibly the output sequence with a sequence of arbitrary *characters*. A suitable memory buffer that will provide storage for the sequences shall be provided by the application as a parameter to a `basic_membuf` constructor. The buffer shall exists for the whole lifetime of the `basic_membuf` object. Every read operation on the input sequence shall be performed by reading the buffer contents. Every write operation on the output sequence shall be immediately written to the buffer. If the program modifies the contents of the buffer by directly accessing it, the behaviour is unspecified.

2 If member function `setbuf()` is called then all requirements in the previous paragraph are immediately relieved from the current buffer and transferred to the new buffer specified by the call.

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

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

4 In additions to the required signatures, implementations are encouraged, but not required, to provide optimized implementations of virtual functions `xsgetn()` and `xsputn()` (27.5.2.4.3 and 27.5.2.4.5 resp.). Any such implementations shall copy buffer elements using `traits::copy()`.

3.2.1.3 basic_membuf constructors [lib.membuf.cons]

```
basic_membuf(charT* s, streamsize n,
    ios_base::openmode which = ios_base::in | ios_base::out);
```

1 **Requires** *s* is a valid pointer to an array of at least *n* elements, *n* `<= max_size()`.

2 **Throws** `invalid_argument` if *s* is a null pointer, `lenght_error` if *n* `> max_size()`.

3 **Effects** Constructs an object of class `basic_membuf`, initializing the base class with `basic_streambuf()` (27.5.2.1), and initializing `mode` with *which*. Initializes the underlying sequence with the buffer starting at *s* and *n* elements long. If *which* `& ios_base::out` is `true`, initializes the output sequence with the underlying sequence. If *which* `& ios_base::in` is `true`, initializes the input sequence with the underlying sequence.

```
basic_membuf(const charT* s, streamsize n,
    ios_base::openmode which = ios_base::in);
```

4 **Requires** *s* is a valid pointer to an array of at least *n* elements, *n* `< max_size()`.

5 **Throws** `invalid_argument` if *s* is a null pointer or *which* `& ios_base::out` is `true`, `lenght_error` if *n* `> max_size()`.

6 **Effects** Constructs an object of class `basic_membuf`, initializing the base class with `basic_streambuf()` (27.5.2.1), and initializing `mode` with *which*. Initializes the underlying sequence with the buffer starting at *s* and *n* elements long with the underlying sequence[3)]. If *which* `& ios_base::in` is `true`, initializes the input sequence.

3.2.1.4 Capacity [lib.membuf.capacity]

```
static streamsize max_size() const;
```

1 **Returns** the maximum allowed size for a buffer that can be managed by a `basic_membuf` object.

3.2.1.5 Overridden virtual functions [lib.membuf.virtuals]

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

1 **Effects** Alters the stream position within one of the controlled sequences, if possible. Effects are identical to those prescribed for function `basic_stringbuf::seekoff()` (27.7.1.3).

2 **Returns** Returned value is identical that of function `basic_stringbuf::seekoff()` (27.7.1.3).

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

3 **Effects** Alters the stream position within the controlled sequences, if possible. Effects are identical to those prescribed for function `basic_stringbuf::seekpos()` (27.7.1.3).

4 **Returns** Returned value is identical that of function `basic_stringbuf::seekpos()` (27.7.1.3).

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

5 **Requires** *s* is a valid pointer to an array of at least *n* elements, with *n* `<= max_size()`

6 **Throws** `invalid_argument` if *s* is a null pointer, `length_error` if *n* `> max_size()`

7 **Effects** Initializes the underlying sequence with the buffer starting at *s* and *n* elements long. If `mode & ios_base::out` is `true`, initializes the output sequence with the new underlying sequence. If `mode & ios_base::in` is `true`, initializes the input sequence with the new underlying sequence.

8 **Returns** this.

3.2.2 Class template basic_imemstream [lib.imemstream]

```
namespace std {
```

[3)]This operation may require casting away the const-ness of *s*. It's responsibility of the implementation to ensure that no write operation is ever attempted on a derefenced pointer obtained by *s*.

```
    template <class charT, class traits = char_traits<charT> >
    class basic_imemstream : public basic_istream<charT,traits> {
    public:
        typedef charT char_type;
        typedef typename traits::int_type int_type;
        typedef typename traits::pos_type pos_type;
        typedef typename traits::off_type off_type;
        typedef traits traits_type;

        // Constructors:
        basic_imemstream(const charT* s, streamsize n,
            ios_base::openmode which = ios_base::in);

        // Members:
        basic_membuf<charT,traits>* rdbuf() const;

    private:
        // basic_membuf<charT,traits> sb; exposition only
    };
}
```

1 The class basic_imemstream supports reading from memory buffers. It uses a basic_membuf object to manage access to the storage. For the sake of exposition, the maintained data is presented here as:

— *sb* the basic_membuf object.

3.2.2.1 basic_imemstream constructors [lib.imemstream.cons]

```
basic_imemstream(const charT* s, streamsize n,
    ios_base::openmode which = ios_base::in);
```

1 **Effects** Constructs an object of class basic_imemstream, initializing the base class with basic_istream(&sb) and initializing sb with basic_membuf(*s*, *n*, *which* | ios_base::in) (27.9.1.1).

3.2.2.2 Member functions [lib.imemstream.members]

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

1 **Returns** &sb.

3.2.3 Class basic_omemstream [lib.omemstream]

```
namespace std {
    template <class charT, class traits = char_traits<charT> >
    class basic_omemstream : public basic_ostream<charT,traits> {
    public:
        // Types:
        typedef charT char_type;
        typedef typename traits::int_type int_type;
        typedef typename traits::pos_type pos_type;
        typedef typename traits::off_type off_type;
        typedef traits traits_type;

        // Constructors:
        basic_omemstream(charT* s, streamsize n,
            ios_base::openmode which = ios_base::out);

        // Members:
        basic_membuf<charT,traits>* rdbuf() const;
    private:
        // basic_membuf<charT,traits> sb; exposition only
    };
}
```

1 The class basic_omemstream supports writing to memory buffers. It uses a basic_membuf object to manage access to the storage. For the sake of exposition, the maintained data is presented here as:

— sb the basic_membuf object.

3.2.3.1 basic_omemstream constructors [lib.omemstream.cons]

```
basic_omemstream(charT* s, streamsize n,
    ios_base::openmode which = ios_base::out);
```

1 **Effects** Constructs an object of class basic_omemstream, initializing the base class with basic_ostream(&sb) and initializing sb with basic_membuf(*s*, *n*, *which* | ios_base::out) (27.9.1.1).

3.2.3.2 Member functions [lib.omemstream.members]

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

1 **Returns** &sb.

3.2.4 Class template basic_memstream [lib.memstream]

```
namespace std {
    template <class charT, class traits = char_traits<charT> >
    class basic_memstream : public basic_iostream<charT,traits> {
    public:
        // Types:
        typedef charT char_type;
        typedef typename traits::int_type int_type;
        typedef typename traits::pos_type pos_type;
        typedef typename traits::off_type off_type;
        typedef traits traits_type;

        // Constructors:
        basic_memstream(charT* s, streamsize n,
            ios_base::openmode which = ios_base::out | ios_base::in);

        // Members:
        basic_membuf<charT,traits>* rdbuf() const;

    private:
        // basic_membuf<charT,traits> sb; exposition only
    };
}
```

1 The class template basic_memstream supports reading and writing from/to memory buffers. It uses a basic_membuf object to manage access to the storage. For the sake of exposition, the maintained data is presented here as

— sb the basic_membuf object.

3.2.4.1 basic_memstream constructors [lib.memstream.cons]

```
basic_memstream(charT* s, streamsize n,
    ios_base::openmode which = ios_base::out | ios_base::in);
```

1 **Effects** Constructs an object of class `basic_memstream`, initializing the base class with `basic_iostream(&sb)` and initializing `sb` with `basic_membuf(s, n, which)`.

3.2.4.2 Member functions [lib.memstream.members]

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

1 **Returns** `&sb`.

4 Remarks

In principle, it should be possible to use as target/source whatever buffer the user is able to allocate. Unfortunately, the only portable way to reposition the output sequence is by using function `basic_streambuf::pbump()`. As such function takes an `int` parameter, it effectively limits the size of the buffer to `numeric_limits<int>::max()`. This issue is closely related with LWG issue #255. Function `basic_membuf::max_size()` has been introduced for the sole purpose to allow implementations based only on the current standard, otherwise it should not be necessary. See also the comments in functions `setgpos()` and `setppos()` in the reference implementation code. Notice that the problem only arises for the output sequence, as the input sequence can be repositioned with `basic_streambuf::setg()`, which does not suffer the limitations of `basic_streambuf::gbump()`.

5 Unresolved issues

The following issues have been raised and not yet addressed:

a) About the `basic_membuf` constructor that takes a const pointer, the proposed text specify that an exception is thrown if `which & std::ios_base::out` is `true`. Other solutions could be considered, such as silently ignoring the case by assigning `mode = which & ~std::ios_base::out`.

b) Should the buffer and possibly the stream templates provide an accessor to the underlying buffer? In that case the name should avoid any reference to strings. Among possible names, `data`, with obvious analogy with `basic_string::data`, seems a good candidate.

c) Should the buffer and possibly the stream templates provide an accessor to the underlying size? In the reference implementation that would be possibile with little effort. The most natural name for such accessor would be `size`.

d) The buffer and possibly the stream templates could provide a constructor that takes a range in addition or instead of the proposed pointer/size approach.

e) If the buffer contents are directly modified by the program then the behaviour is said to be unspecified. Should it be specified instead? The two other options (implementation-defined and undefined) don't look very attractive.

6 Acknowledgements

The author would like to thank Howard Hinnant for his encouragement and support.

Annex A
(informative)
Reference implementation

```
#ifndef INCLUDED_MEMSTREAM_HPP
#define INCLUDED_MEMSTREAM_HPP

#include <istream>
#include <ostream>
#include <streambuf>
#include <stdexcept>
#include <limits>

template <class charT, class traits = std::char_traits<charT> >
class basic_membuf
  : public std::basic_streambuf<charT, traits>
{
public:
  typedef std::basic_streambuf<charT, traits> base_type;
  typedef typename traits::int_type int_type;
  typedef typename traits::pos_type pos_type;
  typedef typename traits::off_type off_type;

  basic_membuf(charT* s, std::streamsize n,
    std::ios_base::openmode mode = std::ios_base::in | std::ios_base::out)
  {
    if (!s)
      throw std::invalid_argument("null-pointer␣not␣allowed");
    if(n > max_size())
      throw std::length_error("buffer␣size␣too␣large");;

    bufsize_ = n;
    if (mode & std::ios_base::out)
      this->setp(s, s + n);
    if (mode & std::ios_base::in)
      this->setg(s, s, s + n);
  }

  basic_membuf(const charT* s, std::streamsize n,
    std::ios_base::openmode mode = std::ios_base::in)
  {
    if (!s)
      throw std::invalid_argument("null-pointer␣not␣allowed");
    if ((mode & std::ios_base::out) != 0)
      throw std::invalid_argument("std::ios_base::out␣flag␣not␣allowed");
    if(n > max_size())
      throw std::length_error("buffer␣size␣too␣large");

    bufsize_ = n;
    if (mode & std::ios_base::in)
    {
      charT* muts = const_cast<charT*>(s);
      this->setg(muts, muts, muts + n);
    }
  }

  static std::streamsize max_size()
  {
    // see setgpos/setppos
    return static_cast<std::streamsize>(std::numeric_limits<int>::max());
  }

protected:
  virtual pos_type seekoff(
    off_type off,
    std::ios_base::seekdir way,
    std::ios_base::openmode which = std::ios_base::in | std::ios_base::out)
  {
    if ((which & (std::ios_base::in | std::ios_base::out))
      == (std::ios_base::in | std::ios_base::out)
      && this->gptr() && this->pptr())
    {
      // reposition both sequences
      switch (way)
      {
      case std::ios_base::beg:
        break;

      case std::ios_base::cur:
        off = -1; // this case is not allowed
        break;

      case std::ios_base::end:
        off += bufsize_;
        break;
      }

      if (off >= 0 && off <= bufsize_)
      {
        setgpos(off);
        setppos(off);
      }
      else
      {
        off = -1;
      }
    }
    else if ((which & std::ios_base::in) && this->gptr())
    {
      switch (way)
      {
      case std::ios_base::beg:
        break;

      case std::ios_base::cur:
        off += this->gptr() - this->eback();
        break;

      case std::ios_base::end:
        off += bufsize_;
        break;
      }

      if (off >= 0 && off <= bufsize_)
      {
        setgpos(off);
      }
      else
      {
        off = -1;
      }
    }
    else if ((which & std::ios_base::out) && this->pptr())
    {
      switch (way)
      {
      case std::ios_base::beg:
        break;

      case std::ios_base::cur:
        off += this->pptr() - this->pbase();
        break;

      case std::ios_base::end:
        off += bufsize_;
        break;
      }

      if (off >= 0 && off <= bufsize_)
      {
        setppos(off);
      }
      else
      {
        off = -1;
      }
    }
    else
    {
      // no sequence can be repositioned
      off = -1;
    }

    return pos_type(off);
  }

  virtual pos_type seekpos(
    pos_type sp,
    std::ios_base::openmode which = std::ios_base::in | std::ios_base::out)
  {
    bool moved = false;

    off_type off(sp);
    if (off >= 0 && off <= bufsize_)
    {
      if ((which & std::ios_base::in) && this->gptr())
      {
        setgpos(off);
        moved = true;
      }

      if ((which & std::ios_base::out) && this->pptr())
      {
        setppos(off);
        moved = true;
      }
    }

    return moved ? pos_type(off) : pos_type(off_type(-1));
  }

  virtual std::streamsize xsgetn(char_type* s, std::streamsize n)
  {
    if (this->gptr())
    {
      std::streamsize pos = this->gptr() - this->eback();
      n = std::min(n, bufsize_ - pos);
      traits::copy(s, this->gptr(), n);
      setgpos(pos + n); // deliberately preferring setgpos to gbump
    }
    else
    {
      n = 0;
    }
    return n;
  }

  virtual std::streamsize xsputn(const char_type* s, std::streamsize n)
  {
    if (this->pptr())
    {
      std::streamsize pos = this->pptr() - this->pbase();
      n = std::min(n, bufsize_ - pos);
      traits::copy(this->pptr(), s, n);
      setppos(pos + n); // deliberately preferring setppos to pbump
    }
    else
    {
      n = 0;
    }

    return n;
  }

   virtual base_type* setbuf(char_type* s, std::streamsize n)
  {
    if (!s)
      throw std::invalid_argument("null-pointer␣not␣allowed");
    if(n > max_size())
      throw std::length_error("buffer␣size␣too␣large");;

    bufsize_ = n;
    if (this->pptr())
      this->setp(s, s + n);
    if (this->gptr())
      this->setg(s, s, s + n);
    return this;
  }

private:
  std::streamsize bufsize_;

  void setgpos(std::streamsize pos)
  {
    // these casts are required because gbump() takes an int argument
    // the conversions are ok because max_size() == numeric_limits<int>::max()
    this->gbump(static_cast<int>(pos)
                - static_cast<int>(this->gptr() - this->eback()));

    // alternatively this function could be written as:
    // this->setg(this->eback(), this->eback() + pos, this->egptr());
    // notice that this form would not suffer the max_size() requirement
  }

  void setppos(std::streamsize pos)
  {
    // these casts are required because pbump() takes an int argument
    // the conversions are ok because max_size() == numeric_limits<int>::max()
    this->pbump(static_cast<int>(pos)
                - static_cast<int>(this->pptr() - this->pbase()));

    // to avoid the max_size() requirement we would need some other way
    // to reposition the put pointer. For example, the Dinkumware C++ library
    // provides a non-standard setp() with three arguments that would make it
    // possible to write:
    // this->setp(this->pbase(), this->pbase() + pos, this->epptr());
  }
};
```

```
template<class charT, class traits = std::char_traits<charT> >
class basic_imemstream
  : public std::basic_istream<charT, traits>
{
  typedef basic_membuf<charT, traits> buffer_type;

public:
  basic_imemstream(const charT* s, std::streamsize n,
    std::ios_base::openmode mode = std::ios_base::in)
    : std::basic_istream<charT, traits>(0)
    , buffer_(s, n, mode | std::ios_base::in)
  {
    this->init(&buffer_);
  }

  basic_imemstream(const charT* begin, const charT* end,
    std::ios_base::openmode mode = std::ios_base::in)
    : std::basic_istream<charT, traits>(0)
    , buffer_(begin, end - begin, mode | std::ios_base::in)
  {
    this->init(&buffer_);
  }

  buffer_type* rdbuf() const
  {
    return &buffer_;
  }

private:
  buffer_type buffer_;
};

template<class charT, class traits = std::char_traits<charT> >
class basic_omemstream
  : public std::basic_ostream<charT, traits>
{
  typedef basic_membuf<charT, traits> buffer_type;

public:
  basic_omemstream(charT* s, std::streamsize n,
    std::ios_base::openmode mode = std::ios_base::out)
    : std::basic_ostream<charT, traits>(0)
    , buffer_(s, n, mode | std::ios_base::out)
  {
    this->init(&buffer_);
  }

  basic_omemstream(charT* begin, charT* end,
    std::ios_base::openmode mode = std::ios_base::out)
    : std::basic_ostream<charT, traits>(0)
    , buffer_(begin, end - begin, mode | std::ios_base::out)
  {
    this->init(&buffer_);
  }

  buffer_type* rdbuf() const
  {
    return &buffer_;
  }

private:
  buffer_type buffer_;
};

template<class charT, class traits = std::char_traits<charT> >
```

```
class basic_memstream
  : public std::basic_iostream<charT, traits>
{
  typedef basic_membuf<charT, traits> buffer_type;

public:
  basic_memstream(charT* s, std::streamsize n,
    std::ios_base::openmode mode = std::ios_base::in | std::ios_base::out)
    : std::basic_iostream<charT, traits>(0)
    , buffer_(s, n, mode)
  {
    this->init(&buffer_);
  }

  basic_memstream(charT* begin, charT* end,
    std::ios_base::openmode mode = std::ios_base::out)
    : std::basic_iostream<charT, traits>(0)
    , buffer_(begin, end - begin, mode)
  {
    this->init(&buffer_);
  }

  buffer_type* rdbuf() const
  {
    return &buffer_;
  }

private:
  buffer_type buffer_;
};

typedef basic_membuf<char>         membuf;
typedef basic_imemstream<char>     imemstream;
typedef basic_omemstream<char>     omemstream;
typedef basic_memstream<char>      memstream;

typedef basic_membuf<wchar_t>      wmembuf;
typedef basic_imemstream<wchar_t>  wimemstream;
typedef basic_omemstream<wchar_t>  womemstream;
typedef basic_memstream<wchar_t>   wmemstream;

#endif // INCLUDED_MEMSTREAM_HPP
```