Document number: P0059R1
Date: 2016-02-09
Reply-to: Guy Davidson, guy@hatcat.com
Reply-to: Arthur O'Dwyer, arthur.j.odwyer@gmail.com
Audience: Library Evolution (LEWG), Game dev and low latency (SG14)

A proposal to add a ring span to the standard library

0. Contents

1. Introduction

This proposal introduces a ring to the standard library operating on a span, named ring_span. The ring_span offers similar facilities to std::queue with the additional feature of storing the elements in contiguous memory and being of a fixed size. It is an update to P0059R1. The significant changes are:

1. Making the ring a span rather than an adaptor.
2. Parameterising pop behaviour.
3. Making the try_ functions non-member functions.
4. The addition of an iterator.
5. Eliminating the fixed and dynamic rings, replacing them with a single ring type.

Much of this work was completed with the help of Arthur O'Dwyer. The authors seek feedback on the design of the ring before submitting wording for the standard.

2. Motivation

Queues are widely used containers for collecting data prior to processing in order of entry to the queue (first in, first out). The std::queue container adaptor acts as a wrapper to an underlying container, typically std::deque or std::list. These containers are non-contiguous, which means that each item that is added to a std::queue may prompt an allocation, which will lead to memory fragmentation. The ring_span operates on elements in contiguous non-owned memory, so memory allocation is eliminated.

3. Impact on the standard

This proposal is a pure library extension. It does not require changes to any standard classes, functions or headers.

4. Design decisions

Naming

In the previous version of this paper the name ring_buffer was proposed, but given the new implementation the proposed name is ring_span.

Look like std::queue

There is already an object that offers FIFO support: the std::queue container. The queue grows to accommodate new entries, allocating new memory as necessary. The ring interface can therefore be similar to that of the queue with the addition of try_push, try_emplace and try_pop functions: these must now fail if they are called when the ring is full (or empty in the case of try_pop), and should therefore signal that condition by returning a success/fail value.

push and pop

Pushing items is a simple matter of assigning to a pre-existing element. Popping items is a more complicated matter than in other containers. If an item is popped from a std::queue it is destroyed and the memory is released. In the case of a ring_span however, it does not own the memory so a different strategy must be pursued. There are three things that could happen when an object is popped from a ring_span, besides the usual container housekeeping:

1. The object is destroyed via the class destructor and the memory is left in an undefined state.
2. The object is replaced with a default-constructed object.
3. The object is replaced with a copy of a user-specified object.

This is a choice that will depend on the type being contained. For example, if the type is not default-constructible, option 2 is unavailable. If the type is not assignable, options 2 and 3 are unavailable. There is no single solution that covers all these situations, so as part of the definition of ring_span a number of pop strategy objects are defined. A strategy can be chosen at the point of declaration of an instance of a ring_span as a template parameter.

5. Header <ring_span> synopsis

This section contains the header declarations. Example definitions are also provided for clarity and to aid specification of the definitions.

```
namespace std::experimental {
 template <typename T> struct null_popper {
  void operator()(T&); };
 template <typename T> struct default_popper {
  T operator()(T& t); };
 template <typename T> struct copy_popper {
  copy_popper(T&& t);
  T operator()(T& t);
  T copy; };

 template <typename, bool> class ring_iterator;
```

```
 template<typename T, class Popper = default_popper<T>> class ring_span
{
 public:
  using type = ring_span<T, Popper>;
  using size_type = std::size_t;
  using value_type = T;
  using pointer = T*;
  using reference = T&;
  using const_reference = const T&;
  using iterator = ring_iterator<type, false>;
  using const_iterator = ring_iterator<type, true>;

  friend class ring_iterator<type, false>;
  friend class ring_iterator<type, true>;

  template <class ContiguousIterator>
  ring_span(ContiguousIterator begin,
       ContiguousIterator end,
       Popper p = Popper()) noexcept;
  template <class ContiguousIterator>
  ring_span(ContiguousIterator begin,
       ContiguousIterator end,
       ContiguousIterator first,
       size_type size,
       Popper p = Popper()) noexcept;

  bool empty() const noexcept;
  bool full() const noexcept;
  size_type size() const noexcept;
  size_type capacity() const noexcept;

  reference front() noexcept;
  const_reference front() const noexcept;
  reference back() noexcept;
  const_reference back() const noexcept;

  iterator begin() noexcept;
  const_iterator begin() const noexcept;
  const_iterator cbegin() const noexcept;
  iterator end() noexcept;
  const_iterator end() const noexcept;
  const_iterator cend() const noexcept;

  void push_back(const value_type& from_value)
    noexcept(std::is_nothrow_copy_assignable<T>::value);
```

```
  void push_back(value_type&& from_value)
    noexcept(std::is_nothrow_move_assignable<T>::value);
  template<class... FromType>
  void emplace_back(FromType&&... from_value)
    noexcept(std::is_nothrow_constructible<T, FromType...>::value &&
             std::is_nothrow_move_assignable<T>::value);
  auto pop_front();

  void swap(type& rhs) noexcept;

// Example implementation
 private:
  reference at(size_type idx) noexcept;
  const_reference at(size_type idx) const noexcept;
  size_type back_idx() const noexcept;
  void increase_size() noexcept;

  T* m_data;
  size_type m_size;
  size_type m_capacity;
  size_type m_front_idx;
  Popper m_popper;
 };

 template <typename Ring, bool is_const>
 class ring_iterator {
 public:
  using type = ring_iterator<Ring, is_const>;
  using value_type = typename Ring::value_type;
  using difference_type = std::ptrdiff_t;
  using pointer = typename std::conditional_t<is_const,
                  const value_type, value_type>*;
  using reference = typename std::conditional_t<is_const,
                    const value_type, value_type>&;
  using iterator_category = std::random_access_iterator_tag;

  template <bool C>
  bool operator==(const ring_iterator<Ring, C>& rhs) const noexcept;
  template <bool C>
  bool operator<(const ring_iterator<Ring, C>& rhs) const noexcept;

  reference operator*() const noexcept;
  type& operator++() noexcept;
  type operator++(int) noexcept;
  type& operator--() noexcept;
  type operator--(int) noexcept;
```

```
  friend type& operator+=(type& it, int i) noexcept;
  friend type& operator-=(type& it, int i) noexcept;

// Example implementation
 private:
  friend Ring;
  using size_type = typename Ring::size_type;
  ring_view_iterator(size_type idx, Ring* rv) noexcept;
  size_type modulo_capacity(size_type idx) noexcept;
  size_type m_idx;
  Ring* m_rv;
 };

 template<typename T, class Popper = default_popper<T>>
 bool try_push_back(ring_span<T, Popper>& r, const value_type&
from_value)
  noexcept(std::is_nothrow_copy_assignable<T>::value);

 template<typename T, class Popper = default_popper<T>>
 bool try_push_back(ring_span<T, Popper>& r, value_type&& from_value)
  noexcept(std::is_nothrow_move_assignable<T>::value);

 template<typename T, class Popper = default_popper<T>>
 template<class... FromType>
 bool try_emplace_back(ring_span<T, Popper>& r, FromType&&...
from_value)
  noexcept(std::is_nothrow_constructible<T, FromType...>::value &&
           std::is_nothrow_move_assignable<T>::value);

 template<typename T, class Popper = default_popper<T>>
 auto try_pop_front(ring_span<T, Popper>& r);
```

6. Function specifications: *_popper

The null_popper object does nothing to the item being popped from the ring.

```
 template <typename T>
 void null_popper::operator()(T&)
 {};
```

The default_popper object moves the item being popped from the ring into the return value.

```
 template <typename T>
 T default_popper::operator()(T& t)
 {
  return std:move(t);
 }
```

The copy_popper object replaces the item being popped from the ring with a copy of an item of the contained type, chosen at the declaration site.

```
template <typename T>
copy_popper::copy_popper(T&& t)
  : copy(std::move(t))
{}

template <typename T>
T copy_popper::operator()(T& t)
{
  T old = t;
  t = copy;
  return t;
}
```

7. Function specifications: ring_span

The first constructor takes a range delimited by two contiguous iterators and an instance of a popper. After this constructor is executed, the capacity of the ring is the distance between the two iterators and the size of the ring is its capacity. A typical implementation would be

```
template<typename T, class Popper>
template<class ContiguousIterator>
ring_span<T, Popper>::ring_span(ContiguousIterator begin,
ContiguousIterator end, Popper p) noexcept
  : m_data(&*begin)
  , m_size(end - begin)
  , m_capacity(end - begin)
  , m_front_idx(0)
  , m_popper(std::move(p))
  {}
```

The second constructor creates a partially full ring. It takes a range delimited by two contiguous iterators, a third iterator which points to the oldest item of the ring, a size parameter which indicates how many items are in the ring, and an instance of a popper. After this constructor is executed, the capacity of the ring is the distance between the first two iterators and the size of the ring is the size parameter. A typical implementation would be

```
template<typename T, class Popper>
template<class ContiguousIterator>
ring_span<T, Popper>::ring_span(ContiguousIterator begin,
ContiguousIterator end, ContiguousIterator first, size_type size, Popper
p = Popper()) noexcept
 : m_data(&*begin)
 , m_size(size)
 , m_capacity(end - begin)
 , m_front_idx(first - begin)
 , m_popper(std::move(p))
```

```
{}
```

empty(), full(), size() and capacity() behave as expected. Typical implementations would be:

```
template<typename T, class Popper>
bool ring_span<T, Popper>::empty() const noexcept
{ return m_size == 0; }
template<typename T, class Popper>
bool ring_span<T, Popper>::full() const noexcept
{ return m_size == m_capacity; }
template<typename T, class Popper>
ring_span<T, Popper>::size_type ring_span<T, Popper>::size() const
noexcept
{ return m_size; }
template<typename T, class Popper>
ring_span<T, Popper>::size_type ring_span<T, Popper>::capacity() const
noexcept
{ return m_capacity; }
```

front() and back() return the oldest and newest items in the ring. Typical implementations would be:

```
template<typename T, class Popper>
ring_span<T, Popper>::reference ring_span<T, Popper>::front() noexcept
{ return *begin(); }
template<typename T, class Popper>
ring_span<T, Popper>::reference ring_span<T, Popper>::back() noexcept
{ return *(end() - 1); }
template<typename T, class Popper>
ring_span<T, Popper>::const_reference ring_span<T, Popper>::front()
const noexcept
{ return *begin(); }
template<typename T, class Popper>
ring_span<T, Popper>::const_reference ring_span<T, Popper>::back() const
noexcept
{ return *(end() - 1); }
```

begin(), cbegin(), end() and cend() return iterators to the oldest and one-past-the-newest items. Typical implementations would be:

```
template<typename T, class Popper>
ring_span<T, Popper>::iterator ring_span<T, Popper>::begin() noexcept
{ return iterator(0, this); }
template<typename T, class Popper>
ring_span<T, Popper>::iterator ring_span<T, Popper>::end() noexcept
{ return iterator(size(), this); }
template<typename T, class Popper>
ring_span<T, Popper>::const_iterator ring_span<T, Popper>::begin() const
noexcept
```

```
{ return const_iterator(0, this); }
template<typename T, class Popper>
ring_span<T, Popper>::const_iterator ring_span<T, Popper>::cbegin()
const noexcept
{ return const_iterator(0, this); }
template<typename T, class Popper>
ring_span<T, Popper>::const_iterator ring_span<T, Popper>::end() const
noexcept
{ return const_iterator(size(), this); }
template<typename T, class Popper>
ring_span<T, Popper>::const_iterator ring_span<T, Popper>::cend() const
noexcept
{ return const_iterator(size(), this); }
```

The push_back() functions add an item after the most recently added item. The emplace_back() function creates an item after the most recently added item. If the size of the ring equals the capacity of the ring, then the oldest item is replaced. Otherwise, the size of the ring is increased by one. Typical implementations would be:

```
template<typename T, class Popper>
template<bool b=true, typename=std::enable_if_t<b &&
std::is_copy_assignable<T>::value>>
void ring_span<T, Popper>::push_back(const T& value)
noexcept(std::is_nothrow_copy_assignable<T>::value)
{
 m_data[back_idx()] = value;
 increase_size();
}

template<typename T, class Popper>
template<bool b=true, typename=std::enable_if_t<b &&
std::is_move_assignable<T>::value>>
void ring_span<T, Popper>::push_back(T&& value)
noexcept(std::is_nothrow_move_assignable<T>::value)
{
 m_data[back_idx()] = std::move(value);
 increase_size();
}

template<typename T, class Popper>
template<class... FromType>
void ring_span<T, Popper>::emplace_back(FromType&&... from_value)
noexcept(std::is_nothrow_constructible<T, FromType...>::value &&
         std::is_nothrow_move_assignable<T>::value);
{
 m_data[back_idx()] = T(std::forward<FromType>(from_value)...);
 increase_size();
```

```
}
```

The pop() function checks the size of the ring, asserting if it is zero. If it is non-zero, it passes a reference to the oldest item to the Popper for transformation, reduces the size and advances the front of the ring. By returning the item from pop, we are able to contain smart pointers. A typical implementation might be:

```
template<typename T, class Popper>
auto ring_span<T, Popper>::pop_front()
{
 assert(m_size != 0);
 auto old_front_idx = m_front_idx;
 m_front_idx = (m_front_idx + 1) % m_capacity;
 --m_size;
 return m_popper(m_data[old_front_idx]);
}
```

The swap() function is trivial. A typical implementation might be:

```
template<typename T, class Popper>
void ring_span<T, Popper>::swap(ring_span<T, Popper>& rhs)
noexcept(std::__is_nothrow_swappable<Popper>::value)
{
 using std::swap;
 swap(m_data, rhs.m_data);
 swap(m_size, rhs.m_size);
 swap(m_capacity, rhs.m_capacity);
 swap(m_front_idx, rhs.m_front_idx);
 swap(m_popper, rhs.m_popper);
}
```

For the sake of clarity, the private implementation used to describe these functions is as follows:

```
template<typename T, class Popper>
ring_span<T, Popper>::reference ring_span<T, Popper>::at(size_type i)
noexcept
{ return m_data[(m_front_idx + i) % m_capacity]; }

template<typename T, class Popper>
ring_span<T, Popper>::const_reference ring_span<T, Popper>::at(size_type
i) const noexcept
{ return m_data[(m_front_idx + i) % m_capacity]; }

template<typename T, class Popper>
ring_span<T, Popper>::size_type ring_span<T, Popper>::back_idx() const
noexcept
{ return (m_front_idx + m_size) % m_capacity; }
```

```
template<typename T, class Popper>
void ring_span<T, Popper>::increase_size() noexcept
{ if (++m_size > m_capacity) { m_size = m_capacity; } }
```

8. Function specifications: ring_iterator

The equality and comparison operators use the index of the iterator to compare their order in the ring. The iterators must be constructed from the same ring. They are equivalent if they point to the same item. operator< will return true if the item pointed to by the member is newer than the item pointed to by the parameter. Typical implementations might be:

```
template <typename Ring, bool is_const>
template<bool C>
bool ring_iterator<Ring, is_const>::operator==(const ring_iterator<Ring,
C>& rhs) const noexcept
{ return (modulo_capacity(m_idx) == rhs.modulo_capacity(m_idx)) && (m_rv
== rhs.m_rv); }

template <typename Ring, bool is_const>
template<bool C>
bool ring_iterator<Ring, is_const>::operator<(const ring_iterator<Ring,
C>& rhs) const noexcept
{ return (modulo_capacity(m_idx) < rhs.modulo_capacity(m_idx)) && (m_rv
== rhs.m_rv); }
```

The dereferencing operator and the pre- and post- increment operators are trivial. Typical implementations might be:

```
template <typename Ring, bool is_const>
ring_iterator<Ring, is_const>::reference ring_iterator<Ring,
is_const>::operator*() const noexcept
{ return m_rv->at(m_idx); }

template <typename Ring, bool is_const>
ring_iterator<Ring, is_const>& ring_iterator<Ring,
is_const>::operator++() noexcept
{ ++m_idx; return *this; }

template <typename Ring, bool is_const>
ring_iterator<Ring, is_const> ring_iterator<Ring,
is_const>::operator++(int) noexcept
{ auto r(*this); ++*this; return r; }

template <typename Ring, bool is_const>
ring_iterator<Ring, is_const>& ring_iterator<Ring,
is_const>::operator--() noexcept
{ ++m_idx; return *this; }

template <typename Ring, bool is_const>
```

```
ring_iterator<Ring, is_const> ring_iterator<Ring,
is_const>::operator--(int) noexcept
{ auto r(*this); ++*this; return r; }
```

non-member operator+= and operator-= are also trivial. Typical implementations might be:

```
template <typename Ring, bool is_const>
ring_iterator<Ring, is_const>& operator+=(ring_iterator<Ring, is_const>&
it, int i) noexcept
{ it.m_idx += i; return it; }

template <typename Ring, bool is_const>
ring_iterator<Ring, is_const>& operator-=(ring_iterator<Ring, is_const>&
it, int i) noexcept
{ it.m_idx -= i; return it; }
```

For the sake of clarity, a private constructor might be implemented like this:

```
template <typename Ring, bool is_const>
ring_iterator<Ring, is_const>::ring_iterator(size_type idx, Ring* rv)
noexcept
: m_idx(idx) , m_rv(rv) {}
```

The modulo capacity normalises the index, as required for ordering and equality functions:

```
template <typename Ring, bool is_const>
ring_iterator<Ring, is_const>::size_type ring_iterator<Ring,
is_const>::modulo_capacity(size_type idx)
{
 return idx % m_rv->capacity();
}
```

9. Sample use

```
#include <ring_span>
#include <cassert>

using std::experimental::ring_span;

void ring_test()
{
  std::array<int, 5> buffer;
  auto Q = ring_span<int>(buffer.begin(), buffer.end(), buffer.begin(),
0);

  Q.push_back(7);
  Q.push_back(3);
  assert(Q.size() == 2);
  assert(Q.front() == 7);
```

```
  Q.pop_front();
  assert(Q.size() == 1);

  Q.push_back(18);
  auto Q2 = Q;
  assert(Q2.front() == 3);
  assert(Q2.back() == 18);

  auto Q3 = std::move(Q2);
  assert(Q3.front() == 3);
  assert(Q3.back() == 18);

  auto Q4(Q3);
  assert(Q4.front() == 3);
  assert(Q4.back() == 18);

  auto Q5(std::move(Q3));
  assert(Q5.front() == 3);
  assert(Q5.back() == 18);
  assert(Q5.size() == 2);

  Q5.pop_front();
  Q5.pop_front();
  assert(Q5.empty());

  std::array<int, 5> buffer2;
  auto Q6 = ring_span<int>(buffer2.begin(), buffer2.end(),
buffer2.begin(), 0);
  Q6.push_back(6);
  Q6.push_back(7);
  Q6.push_back(8);
  Q6.push_back(9);
  Q6.emplace(10);
  Q6.swap(Q5);
  assert(Q6.empty());
  assert(Q5.size() == 5);
  assert(Q5.front() == 6);
  assert(Q5.back() == 10);
}
```

The most obvious use for the ring queue would be for communicating between threads:

```
void sg14_test::thread_communication_test()
{
  std::array<int, 10> buffer;
  std::mutex m;
  std::condition_variable cv;
```

```
  auto r = ring_span<int>(buffer.begin(), buffer.end(), buffer.begin(),
0);

  auto ci = std::async(std::launch::async, [&]()
  {
    int val = 0;
    do
    {
      std::cin >> val;
      {
        std::lock_guard<std::mutex> lg(m);
        r.push_back(val);
        cv.notify_one();
      }
    } while (val != -1);
  });

  auto po = std::async(std::launch::async, [&]()
  {
    int val = 0;
    do
    {
      std::unique_lock<std::mutex> lk(m);
      cv.wait(lk);
      val = r.pop();
      std::cout << val << std::endl;
      lk.unlock();
    } while (val != -1);
  });
}
```

10. Future work

n3353 describes a proposal for a concurrent queue. The interface is quite different from ring. A concurrent ring could be adapted from the interface specified therein should n3353 be accepted into the standard.

The popper class templates are defined at an overly broad scope, rather than in the scope of the ring_span. However, no way of doing this is immediately apparent, beyond the obvious solution of creating a ring namespace and defining the poppers, the span, the iterator and the try_* functions inside it. Since this is somewhat counterintuitive in the context of the remainder of the standard library, the authors remain open to suggestions. If the popper class templates might have use in other container spans, then they could remain in the broader scope.

11. Acknowledgements

Thanks to Jonathan Wakely for sprucing up the first draft of the ring interface.

Thanks to the SG14 forum contributors: Nicolas Guillemot, John McFarlane, Scott Wardle, Chris Gascoyne, Matt Newport.
Thanks to the SG14 meeting contributors: Charles Beattie, Brent Friedman, Billy Baker, Bob, Arthur O'Dwyer, Sean Middleditch, Ville Voutilainen.
Thanks to Michael McLaughlin for commentary on the draft of the text.
Thanks to Arthur O'Dwyer for unifying the interfaces of the original rings into a single span.
Thanks to Lawrence Crowl for pointing me to his paper on concurrent queues, n3353.
Special thanks also to Michael Wong for convening and shepherding SG14.