Document number: D0059R2
Date: 2015-09-21
Project: Programming Language C++, LEWG, SG14
Reply-to: Guy Davidson, guy@hatcat.com

A proposal to add rings to the standard library

Introduction

This proposal introduces a ring adaptor suitable for adapting arrays and vectors for use as fixed size queues.

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, deque or list. These containers are non-contiguous, which means that each item that is added to the queue may prompt an allocation, which will lead to memory fragmentation.

The ring is an adaptor offering the same facilities as the queue adaptor with the additional feature of storing the elements in contiguous memory, minimising the incidence and amount of memory allocation.

Impact on the standard

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

Design decisions

Naming

The subject of naming this entity has been covered in private mails and on the SG14 reflector. There are several candidates. The most obvious one is circular_buffer, but such an object exists in Boost; it is bidirectional and supports random access iterators, which are unnecessary for a FIFO structure. Rolling_queue was considered, but rolling seems like an unusual prefix. Finally, there was cyclic_buffer, ring_buffer and ring. As this entity is an *adaptor* for a buffer in the form of an array or a vector, rather than a buffer itself, the buffer suffix seemed inappropriate, thus the name ring was the last candidate standing.

Look like std::queue

There is already an adaptor that offers FIFO support, std::queue. The queue grows to accommodate new entries, allocating new memory as necessary. For ease of use the ring interface should match this one as far as appropriate.

The interface for std::queue allows for unlimited addition of elements, which is not appropriate for a ring. If elements are to remain contiguous it is impractical to allow a ring to grow since this would involve moving the elements to another address in memory. Therefore, the size of the ring needs to be decided either at compile time or at instantiation time.

The ring interface can therefore be similar to that of the queue with the exception of the push and emplace functions: these must now fail if they are called when the ring is full, and should therefore signal that by returning a success/fail value.

Adapt existing containers

There are two contiguous memory containers that can already be adapted to this purpose, std::array and std::vector. The array would be used when the cost of default-construction of T is acceptable and the size of the queue is known at compile time. If one of those constraints is not met, then the vector would be used. These shall be called static_ring and dynamic_ring. However, the option to use other containers defined by the user that satisfy the appropriate constraints remains open.

Technical specifications

Header <ring> synopsis

```
namespace std {
 template<typename T, std::size_t capacity> class static_ring {
 public:
  typedef std::array<T, capacity> container_type;
  typedef typename container_type::value_type value_type;
  typedef typename container_type::size_type size_type;
  typedef typename container_type::reference reference;
  typedef typename container_type::const_reference const_reference;
  typedef typename container_type::iterator iterator;
  typedef typename container_type::const_iterator const_iterator;
  typedef typename container_type::reverse_iterator reverse_iterator;
  typedef typename container_type::const_reverse_iterator
const_reverse_iterator;

  static_ring()
noexcept(std::is_nothrow_default_constructible<T>::value);
  static_ring(const static_ring& rhs)
noexcept(std::is_nothrow_copy_constructible<T>::value);
  static_ring(static_ring&& rhs)
noexcept(std::is_nothrow_move_constructible<T>::value);
static_ring(const container_type& rhs)
noexcept(std::is_nothrow_copy_constructible<T>::value);
static_ring(container_type&& rhs)
noexcept(std::is_nothrow_move_constructible<T>::value);
  static_ring& operator=(const static_ring& rhs)
noexcept(std::is_nothrow_copy_assignable<T>::value);
  static_ring& operator=(static_ring&& rhs)
noexcept(std::is_nothrow_move_assignable<T>::value);
  bool push(const value_type& from_value)
noexcept(std::is_nothrow_copy_assignable<T>::value);
  bool push(value_type&& from_value)
noexcept(std::is_nothrow_move_assignable<T>::value);
```

```
  template<class... FromType> bool emplace(FromType&&... from_value)
noexcept(std::is_nothrow_constructible<T, FromType...>::value &&
std::is_nothrow_move_assignable<T>::value);
  void pop();
  bool empty() const noexcept;
  size_type size() const noexcept;
  reference front() noexcept;
  const_reference front() const noexcept;
  reference back() noexcept;
  const_reference back() const noexcept;
  void swap(static_ring& rhs) noexcept;

 protected:
  container_type c;
  size_t count;
  iterator next_element;
  iterator last_element;
 };

 template<typename T, class Container = std::vector<T,
std::allocator<T>>> class dynamic_ring {
 public:
  typedef Container container_type;
  typedef typename container_type::value_type value_type;
  typedef typename container_type::size_type size_type;
  typedef typename container_type::reference reference;
  typedef typename container_type::const_reference const_reference;
  typedef typename container_type::iterator iterator;
  typedef typename container_type::const_iterator const_iterator;
  typedef typename container_type::reverse_iterator reverse_iterator;
  typedef typename container_type::const_reverse_iterator
const_reverse_iterator;

  explicit dynamic_ring(size_type initial_capacity);
  dynamic_ring(const dynamic_ring& rhs)
  dynamic_ring(dynamic_ring&& rhs);
  template<typename Alloc> explicit dynamic_ring(const Alloc&);
  template<typename Alloc> dynamic_ring(const dynamic_ring&, const
Alloc&);
  template<typename Alloc> dynamic_ring(dynamic_ring&&, const Alloc&);
  template<typename Alloc> dynamic_ring(const container_type&, const
Alloc&);
  template<typename Alloc> dynamic_ring(container_type&&, const Alloc&);
  dynamic_ring& operator=(const dynamic_ring& rhs);
  dynamic_ring& operator=(dynamic_ring&& rhs);
  bool push(const value_type& from_value);
  bool push(value_type&& from_value);
  template<class... FromType> bool emplace(FromType&&... from_value);
  void pop();
  bool empty() const noexcept;
  size_type size() const noexcept;
  reference front() noexcept;
  const_reference front() const noexcept;
  reference back() noexcept;
  const_reference back() const noexcept;
  void swap(dynamic_ring& rhs) noexcept;

 protected:
  container_type c;
  size_t count;
  iterator next_element;
  iterator last_element;
 };
}
```

Acknowledgements

Thanks to Jonathan Wakely for sprucing up the first draft of the static_ring interface.
Thanks to the SG14 forum contributors: Nicolas Guillemot, John McFarlane, Scott Wardle, Chris Gascoyne, Matt Newport.
Thanks also to Michael Wong for convening and shepherding SG14.