Document number: D0059R0
Date: 2015-09-11
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 adapter suitable for adapting arrays and vectors for use as fixed size queues that are optionally thread-safe.

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 adapter 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 will prompt an allocation, which will lead to memory fragmentation.

The ring is an adapter offering the same facilities as the queue adapter with the additional feature of storing the elements in contiguous memory, minimising the incidence and amount of memory allocation. Also, since a common use for queues is inter-thread communication, the ring offers optional thread safety, a feature the queue was unable to offer when it was introduced.

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. 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 seems inappropriate, thus the name ring was the last candidate standing.

Look like std::queue

There is already an adapter that offers queue support. The queue grows to accommodate new entries, allocating new memory as necessary. This is not an option for a std::array adapter since the size of a std::array object is fixed at compile time.

The interface for std::queue allows for unlimited addition of elements, which is not appropriate for a ring. The ring interface can therefore be identical to that of the queue with

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

Besides this difference, the interfaces for ring and std::queue are identical.

Set the size at compile time or at run time

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& 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 = 8);
  dynamic_ring();
  explicit dynamic_ring(const Allocator& alloc);
  explicit dynamic_ring(size_type count);
  explicit dynamic_ring(size_type count, const Allocator& alloc =
Allocator());
  dynamic_ring(const dynamic_ring& rhs);
  dynamic_ring(dynamic_ring&& rhs);
  dynamic_ring& operator=(const dynamic_ring& rhs);
  dynamic_ring& operator=(dynamic_ring&& rhs);
  void push(const value_type& from_value);
  void push(value_type&& from_value);
  template<class... FromType> void 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;
 };
}
```

Future Issues

Thread safety

It is possible to optionally make the adapter thread-safe through a policy which evaluates to a no-op if thread safety is not required.

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 starting and shepherding SG14.