Document number: D0059R3
Date: 2015-09-23
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. The ring is an adaptor offering the same facilities as the queue adaptor with the additional feature of storing the elements in contiguous memory

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 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.

As stated in the introduction, the ring adapter stores 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 somewhat heavyweight, being bidirectional and supporting 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, although ring_queue has, very recently, also been suggested as a candidate.

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 addition of try_push and try_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 decision tradeoff to be made when deciding which to use is:

1. Is it known how many elements this ring should contain at compile time?
2. Is default-constructing that many elements at instantiation sufficiently cheap?

If both of those questions can be answered in the affirmative, then the std::array should be used. Otherwise, the std::vector should be used, at the cost of an allocation at runtime. The vector will not grow so there will be no further allocations.

This leads to the need for two separate ring classes: one class declares a template parameter for the size of the ring, while the other class takes the size as a constructor parameter. These shall be called fixed_ring and dynamic_ring.

Adapt custom containers

By declaring the container as a template parameter it is possible to adapt custom containers for ring use. For example, a container which contains uninitialised memory could be used instead of an array.

Destroy on pop()

Calling pop does not destroy the object. This is different behaviour from std::queue. pop does not eliminate storage: the container is of a fixed size. At some point the ring will be destroyed, which will cause the destruction of all the contained objects; if they have already been destroyed by pop then double-destruction would take place.

Technical specifications

Header <ring> synopsis:

```
namespace std::experimental {
 template<typename T, std::size_t capacity, class Container =
std::array<T, capacity>> class fixed_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;
```

```
  fixed_ring()
noexcept(std::is_nothrow_default_constructible<T>::value);
  fixed_ring(const fixed_ring& rhs)
noexcept(std::is_nothrow_copy_constructible<T>::value);
  fixed_ring(fixed_ring&& rhs)
noexcept(std::is_nothrow_move_constructible<T>::value);
  fixed_ring(const container_type& rhs)
noexcept(std::is_nothrow_copy_constructible<T>::value);
  fixed_ring(container_type&& rhs)
noexcept(std::is_nothrow_move_constructible<T>::value);
  template<class InputIt> fixed_ring(InputIt first, InputIt last);
  fixed_ring& operator=(const fixed_ring& rhs)
noexcept(std::is_nothrow_copy_assignable<T>::value);
  fixed_ring& operator=(fixed_ring&& rhs)
noexcept(std::is_nothrow_move_assignable<T>::value);
  void push(const value_type& from_value)
noexcept(std::is_nothrow_copy_assignable<T>::value);
  void 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);
  bool try_push(const value_type& from_value)
noexcept(std::is_nothrow_copy_assignable<T>::value);
  bool try_push(value_type&& from_value)
noexcept(std::is_nothrow_move_assignable<T>::value);
  template<class... FromType> bool try_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(fixed_ring& rhs) noexcept;
 };

 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<class InputIt> dynamic_ring(InputIt first, InputIt last,
const Alloc&);
  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);
  void push(const value_type& from_value);
  void push(value_type&& from_value);
  template<class... FromType> void emplace(FromType&&... from_value);
  bool try_push(const value_type& from_value);
  bool try_push(value_type&& from_value);
  template<class... FromType> bool try_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;
 };
}
```

Future work

The existence of two separate classes seems unsatisfactory but I am unable to conceive a suitable interface which accepts both array and vector types and conveys the size appropriately: I remain open to suggestions.

n3353 describes a proposal for a concurrent queue. The interface is quite different from ring, and no guarantees about allocations are mentioned. A concurrent ring could be adapted from the interface specified therein should n3353 be accepted into the standard.

Acknowledgements

Thanks to Jonathan Wakely for sprucing up the first draft of the fixed_ring interface.
Thanks to the SG14 forum contributors: Nicolas Guillemot, John McFarlane, Scott Wardle, Chris Gascoyne, Matt Newport.
Thanks to Michael McLaughlin for commentary on the draft of the text.
Thanks to Lawrence Crowl for pointing me to his paper on concurrent queues, n3353.
Thanks also to Michael Wong for convening and shepherding SG14.