Boost.Circular Buffer

Jan Gaspar

Copyright © 2003-2013 Jan Gaspar

Distributed under the Boost Software License, Version 1.0. (See accompanying file LICENSE_1_0.txt or copy at http://www.boost.org/LICENSE_1_0.txt)

Table of Contents

Introduction 2
Circular_buffer example 4
Rationale 5
Implementation 6
More Examples 10
Header Files 13
Modelled Concepts 14
Template Parameters 15
Trac Tickets 16
Release Notes 17
Acknowledgements 18
Documentation Version Info 19
Boost.Circular_buffer C++ Reference 20
    Header <boost/circular_buffer.hpp> 20
    Header <boost/circular_buffer/base.hpp> 21
    Header <boost/circular_buffer/debug.hpp> 65
    Header <boost/circular_buffer/details.hpp> 65
    Header <boost/circular_buffer/space_optimized.hpp> 65
Index 89

Note

A printer-friendly PDF version of this manual is also available.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Introduction

A Circular Buffer.

Description

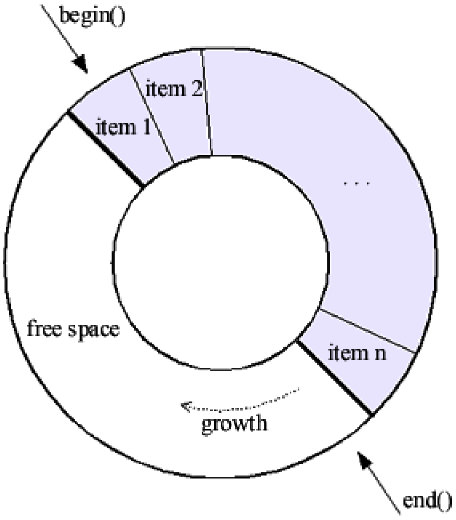
The term circular buffer (also called a *ring* or *cyclic buffer*) refers to an area in memory which is used to store incoming data. When the buffer is filled, new data is written starting at the beginning of the buffer and overwriting the old.

`boost::circular_buffer` is a STL compliant container.

It is a kind of sequence similar to std::list or std::deque. It supports random access iterators, constant time insert and erase operations at the beginning or the end of the buffer and interoperability with std algorithms.

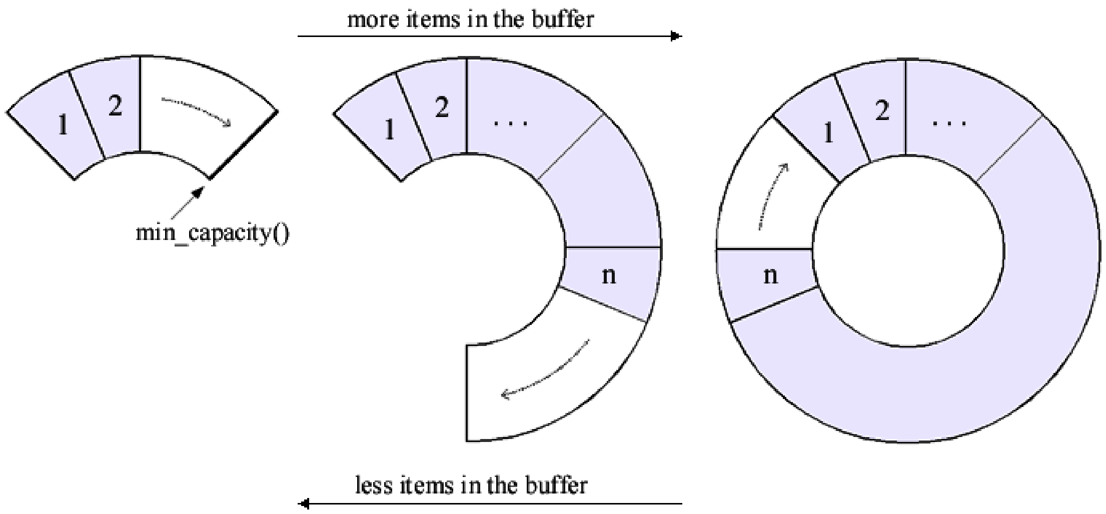
The `circular_buffer` is especially designed to provide **fixed capacity** storage. When its capacity is exhausted, newly inserted elements will cause elements to be overwritten, either at the beginning or end of the buffer (depending on what insert operation is used).

The `circular_buffer` only allocates memory when created, when the capacity is adjusted explicitly, or as necessary to accommodate resizing or assign operations.



There is also a `circular_buffer_space_optimized` version available.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/



`circular_buffer_space_optimized` is an adaptation of the `circular_buffer` which **does not allocate memory all at once when created**, instead it allocates memory as needed.

The predictive memory allocation is similar to typical `std::vector` implementation. Memory is automatically freed as the size of the container decreases.

The memory allocation process of the space-optimized circular buffer. The `min_capacity` of the capacity controller represents the minimal guaranteed amount of allocated memory. The allocated memory will never drop under this value. The default value of the `min_capacity` is set to 0. The `min_capacity` can be set using the constructor parameter `()` `capacity_control` or the function `set_capacity`.

The space-optimized version is, of course, a little slower.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Circular_buffer example

Here is a simple example to introduce the class `circular_buffer`.

For all examples, we need this include:

```
#include <boost/circular_buffer.hpp>
```

This example shows contruction, inserting elements, overwriting and popping.

```
// Create a circular buffer with a capacity for 3 integers.
boost::circular_buffer<int> cb(3);

// Insert threee elements into the buffer.
cb.push_back(1);
cb.push_back(2);
cb.push_back(3);

int a = cb[0];  // a == 1
int b = cb[1];  // b == 2
int c = cb[2];  // c == 3

// The buffer is full now, so pushing subsequent
// elements will overwrite the front-most elements.

cb.push_back(4);  // Overwrite 1 with 4.
cb.push_back(5);  // Overwrite 2 with 5.

// The buffer now contains 3, 4 and 5.
a = cb[0];  // a == 3
b = cb[1];  // b == 4
c = cb[2];  // c == 5

// Elements can be popped from either the front or the back.
cb.pop_back();  // 5 is removed.
cb.pop_front(); // 3 is removed.

// Leaving only one element with value = 4.
int d = cb[0];  // d == 4
```

You can see the full example code at circular_buffer_example.cpp.

The full annotated description is in the C++ Reference section.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Rationale

The basic motivation behind the `circular_buffer` was to create a container which would **work seamlessly with STL**.

Additionally, the design of the `circular_buffer` was guided by the following principles:

- Maximum *efficiency* for envisaged applications.
- Suitable for *general purpose use*.
- The behaviour of the buffer as *intuitive* as possible.
- Suitable for *specialization* by means of adaptors. (The `circular_buffer_space_optimized` is such an example of the adaptor.)
- Easy to *debug*. (See Debug Support for details.)

In order to achieve maximum efficiency, the `circular_buffer` and `circular_buffer_space_optimized` store their elements in a **contiguous region of memory**, which then enables:

- Use of fixed memory and no implicit or unexpected memory allocation.
- Fast constant-time insertion and removal of elements from the front and back.
- Fast constant-time random access of elements.
- Suitability for real-time and performance critical applications.

Possible applications of the circular buffer include:

- Storage of the *most recently received samples*, overwriting the oldest as new samples arrive.
- As an underlying container for a *bounded buffer* (see the Bounded Buffer example, code at circular_buffer_bound_example.cpp).
- A kind of *cache* storing a specified number of last inserted elements.
- Efficient fixed capacity *FIFO (First In, First Out)*,
- Efficient fixed capacity *LIFO (Last In, First Out)* queue which removes the oldest (inserted as first) elements when full.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Implementation

The following paragraphs describe issues that had to be considered during the implementation of the circular_buffer:

Thread-Safety

The thread-safety of the `circular_buffer` is the same as the thread-safety of containers in most STL implementations. This means the `circular_buffer` is not fully thread-safe. The thread-safety is guaranteed only in the sense that simultaneous accesses to distinct instances of the `circular_buffer` are safe, and simultaneous read accesses to a shared `circular_buffer` are safe.

If multiple threads access a single `circular_buffer`, and at least one of the threads may potentially write, then the user is responsible for ensuring mutual exclusion between the threads during the container accesses. The mutual exclusion between the threads can be achieved by wrapping operations of the underlying `circular_buffer` with a lock acquisition and release. (See the Bounded Buffer example code at circular_buffer_bound_example.cpp)

Overwrite Operation

Overwrite operation occurs when an element is inserted into a full `circular_buffer` - the old element is being overwritten by the new one. There was a discussion what exactly "overwriting of an element" means during the formal review. It may be either a destruction of the original element and a consequent inplace construction of a new element or it may be an assignment of a new element into an old one. The `circular_buffer` implements assignment because it is more effective.

From the point of business logic of a stored element, the destruction/construction operation and assignment usually mean the same. However, in very rare cases (if in any) they may differ. If there is a requirement for elements to be destructed/constructed instead of being assigned, consider implementing a wrapper of the element which would implement the assign operator, and store the wrappers instead. It is necessary to note that storing such wrappers has a drawback. The destruction/construction will be invoked on every assignment of the wrapper - not only when a wrapper is being overwritten (when the buffer is full) but also when the stored wrappers are being shifted (e.g. as a result of insertion into the middle of container).

Writing to a Full Buffer

There are several options how to cope if a data source produces more data than can fit in the fixed-sized buffer:

- Inform the data source to wait until there is room in the buffer (e.g. by throwing an overflow exception).
- If the oldest data is the most important, ignore new data from the source until there is room in the buffer again.
- If the latest data is the most important, write over the oldest data.
- Let the producer to be responsible for checking the size of the buffer prior writing into it.

It is apparent that the `circular_buffer` implements the third option. But it may be less apparent it does not implement any other option - especially the first two. One can get an impression that the `circular_buffer` should implement first three options and offer a mechanism of choosing among them. This impression is wrong.

The `circular_buffer` was designed and optimized to be circular (which means overwriting the oldest data when full). If such a controlling mechanism had been enabled, it would just complicate the matters and the usage of the `circular_buffer` would be probably less straightforward.

Moreover, the first two options (and the fourth option as well) do not require the buffer to be circular at all. If there is a need for the first or second option, consider implementing an adaptor of e.g. std::vector. In this case the `circular_buffer` is not suitable for adapting, because, contrary to std::vector, it bears an overhead for its circular behaviour.

Reading/Removing from an Empty Buffer

When reading or removing an element from an empty buffer, the buffer should be able to notify the data consumer (e.g. by throwing underflow exception) that there are no elements stored in it. The `circular_buffer` does not implement such a behaviour for two reasons:

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

- It would introduce a performance overhead.
- No other std container implements it this way.

It is considered to be a bug to read or remove an element (e.g. by calling `front()` or `pop_back()`) from an empty std container and from an empty `circular_buffer` as well. The data consumer has to test if the container is not empty before reading/removing from it by testing `empty()`. However, when reading from the `circular_buffer`, there is an option to rely on the `at()` method which throws an exception when the index is out of range.

Iterator Invalidation

An iterator is usually considered to be invalidated if an element, the iterator pointed to, had been removed or overwritten by another element. This definition is enforced by the Debug Support and is documented for every method. However, some applications utilizing `circular_buffer` may require less strict definition: an iterator is invalid only if it points to an uninitialized memory.

Consider following example:

```
#define BOOST_CB_DISABLE_DEBUG // The Debug Support has to be disabled, otherwise the code pro↵
duces a runtime error.

#include <boost/circular_buffer.hpp>
#include <boost/assert.hpp>
#include <assert.h>

int main(int /*argc*/, char* /*argv*/[])
{

  boost::circular_buffer<int> cb(3);

  cb.push_back(1);
  cb.push_back(2);
  cb.push_back(3);

  boost::circular_buffer<int>::iterator it = cb.begin();

  assert(*it == 1);

  cb.push_back(4);

  assert(*it == 4); // The iterator still points to the initialized memory.

  return 0;
}
```

The iterator does not point to the original element any more (and is considered to be invalid from the "strict" point of view) but it still points to the same valid place in the memory. This "soft" definition of iterator invalidation is supported by the `circular_buffer` but should be considered as an implementation detail rather than a full-fledged feature. The rules when the iterator is still valid can be inferred from the code in soft_iterator_invalidation.cpp.

Move emulation and rvalues

Since Boost 1.54.0 support for move semantics was implemented using the Boost.Move library. If rvalue references are available `circular_buffer` will use them, but if not it uses a close, but imperfect emulation. On such compilers:

- Non-copyable objects can be stored in the containers. They can be constructed in place using `emplace`, or if they support Boost.Move, moved into place.
- The containers themselves are not movable.
- Argument forwarding is not perfect.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

`circular_buffer` will use rvalues and move emulations for value types only if move constructor and move assignment operator of the value type do not throw; or if the value type has no copy constructor.

Some methods won't use move constructor for the value type at all, if the constructor throws. This is required for data consistency and avoidance of situations, when aftrer an exception `circular_buffer` contains moved away objects along with the good ones.

See documentation for `is_copy_constructible`, `is_nothrow_move_assignable` and `is_nothrow_move_constructible` type triats. There you'll find information about how to make constructor of class noexcept and how to make a non-copyable class in C++03 and C++98.

Performance of `circular_buffer` will **greatly improve** if value type has noexcept move constructor and noexcept move assignment.

Exceptions of move_if_noexcept(T&)

Reference documentation of the `circular_buffer` contains notes like "Throws: See Exceptions of `move_if_noexcept(T&)`". That note means the following: `move_if_noexcept(T& value)` does not throws exceptions at all, but it returns `value` as rvalue reference only if class `T` have noexcept move constructor and noexcept move assignment operator; or if it has no copy constructor. Otherwise `move_if_noexcept(T& value)` returns `value` as const reference.

This leads us to the following situation:

- If `value` has a noexcept move constructor and noexcept move assignment operator, then no exceptions will be thrown at all.
- If `value` has a throwing move constructor and some copy constructor, then method may throw exceptions of copy constructor.
- If `value` has no copy constructor, then method may throw exceptions of move constructor.

`move_if_noexcept(T&)` uses Boost.Move, `is_copy_constructible`, `is_nothrow_move_assignable` and `is_nothrow_move_constructible` type triats.

Caveats

The `circular_buffer` should not be used for storing pointers to dynamically allocated objects. When a circular buffer becomes full, further insertion will overwrite the stored pointers - resulting in a **memory leak**. One recommend alternative is the use of smart pointers, for example Boost Smart pointers.

std::auto_ptr

Caution

Any container of `std::auto_ptr` is considered particularly hazardous.

Tip

Never create a circular buffer of `std::auto_ptr`. Refer to Scott Meyers' excellent book Effective STL for a detailed discussion. (Meyers S., Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library. Addison-Wesley, 2001.)

While internals of a `circular_buffer` are circular, **iterators are not**. Iterators of a `circular_buffer` are only valid for the range `\[begin(), end()\]`, so for example: iterators `(begin() - 1)` and `(end() + 1)` are both invalid.

Debug Support

In order to help a programmer to avoid and find common bugs, the `circular_buffer` contains a kind of debug support.

The `circular_buffer` maintains a list of valid iterators. As soon as any element gets destroyed all iterators pointing to this element are removed from this list and explicitly invalidated (an invalidation flag is set). The debug support also consists of many assertions

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

(`BOOST_ASSERT` macros) which ensure the `circular_buffer` and its iterators are used in the correct manner at runtime. In case an invalid iterator is used, the assertion will report an error. The connection of explicit iterator invalidation and assertions makes a very robust debug technique which catches most of the errors.

Moreover, the uninitialized memory allocated by `circular_buffer` is filled with the value `0xcc` in the debug mode. When debugging the code, this can help the programmer to recognize the initialized memory from the uninitialized. For details refer the source code circular_buffer/debug.hpp.

The debug support is enabled only in the debug mode (when the `NDEBUG` is not defined). It can also be explicitly disabled (only for `circular_buffer`) by defining macro BOOST_CB_DISABLE_DEBUG.

Compatibility with Interprocess library

The `circular_buffer` is compatible with the Boost.Interprocess library used for interprocess communication. Considering that the circular_buffer's debug support relies on 'raw' pointers (which is not permited by the Interprocess library) the code has to compiled with `-DBOOST_CB_DISABLE_DEBUG` or `-DNDEBUG` (which disables the Debug Support). Not doing that will cause the compilation to fail.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

More Examples

Summing all the values in a circular buffer

This example shows several functions, including summing all valid values.

```
#include <boost/circular_buffer.hpp>
#include <numeric>
#include <assert.h>

int main(int /*argc*/, char* /*argv*/[])
{
   // Create a circular buffer of capacity 3.
   boost::circular_buffer<int> cb(3);
   assert(cb.capacity() == 3);
   // Check is empty.
   assert(cb.size() == 0);
   assert(cb.empty());

   // Insert some elements into the circular buffer.
   cb.push_back(1);
   cb.push_back(2);

   // Assertions to check push_backs have expected effect.
   assert(cb[0] == 1);
   assert(cb[1] == 2);
   assert(!cb.full());
   assert(cb.size() == 2);
   assert(cb.capacity() == 3);

   // Insert some other elements.
   cb.push_back(3);
   cb.push_back(4);

   // Evaluate the sum of all elements.
   int sum = std::accumulate(cb.begin(), cb.end(), 0);

   // Assertions to check state.
   assert(sum == 9);
   assert(cb[0] == 2);
   assert(cb[1] == 3);
   assert(cb[2] == 4);
   assert(*cb.begin() == 2);
   assert(cb.front() == 2);
   assert(cb.back() == 4);
   assert(cb.full());
   assert(cb.size() == 3);
   assert(cb.capacity() == 3);

   return 0;
}
```

The `circular_buffer` has a capacity of three `int`. Therefore, the size of the buffer will never exceed three. The `std::accumulate` algorithm evaluates the sum of the stored elements. The semantics of the `circular_buffer` can be inferred from the assertions.

You can see the full example code at circular_buffer_sum_example.cpp.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Bounded Buffer Example

The bounded buffer is normally used in a producer-consumer mode: producer threads produce items and store them in the container and consumer threads remove these items and process them. The bounded buffer has to guarantee that

- producers do not insert items into the container when the container is full,
- consumers do not try to remove items when the container is empty,
- each produced item is consumed by exactly one consumer.

This example shows how the `circular_buffer` can be utilized as an underlying container of the bounded buffer.

```
#include <boost/circular_buffer.hpp>
#include <boost/thread/mutex.hpp>
#include <boost/thread/condition.hpp>
#include <boost/thread/thread.hpp>
#include <boost/call_traits.hpp>
#include <boost/bind.hpp>

#include <boost/timer/timer.hpp> // for auto_cpu_timer

template <class T>
class bounded_buffer
{
public:

  typedef boost::circular_buffer<T> container_type;
  typedef typename container_type::size_type size_type;
  typedef typename container_type::value_type value_type;
  typedef typename boost::call_traits<value_type>::param_type param_type;

  explicit bounded_buffer(size_type capacity) : m_unread(0), m_container(capacity) {}

  void push_front(typename boost::call_traits<value_type>::param_type item)
  { // `param_type` represents the "best" way to pass a parameter of type `value_type` to a method.

      boost::mutex::scoped_lock lock(m_mutex);
      m_not_full.wait(lock, boost::bind(&bounded_buffer<value_type>::is_not_full, this));
      m_container.push_front(item);
      ++m_unread;
      lock.unlock();
      m_not_empty.notify_one();
  }

  void pop_back(value_type* pItem) {
      boost::mutex::scoped_lock lock(m_mutex);
      m_not_empty.wait(lock, boost::bind(&bounded_buffer<value_type>::is_not_empty, this));
      *pItem = m_container[--m_unread];
      lock.unlock();
      m_not_full.notify_one();
  }

private:
  bounded_buffer(const bounded_buffer&);              // Disabled copy constructor.
  bounded_buffer& operator = (const bounded_buffer&); // Disabled assign operator.

  bool is_not_empty() const { return m_unread > 0; }
  bool is_not_full() const { return m_unread < m_container.capacity(); }
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
   size_type m_unread;
   container_type m_container;
   boost::mutex m_mutex;
   boost::condition m_not_empty;
   boost::condition m_not_full;
}; //
```

The bounded_buffer relies on Boost.Thread and Boost.Bind libraries and Boost.call_traits utility.

The `push_front()` method is called by the producer thread in order to insert a new item into the buffer. The method locks the mutex and waits until there is a space for the new item. (The mutex is unlocked during the waiting stage and has to be regained when the condition is met.) If there is a space in the buffer available, the execution continues and the method inserts the item at the end of the `circular_buffer`. Then it increments the number of unread items and unlocks the mutex (in case an exception is thrown before the mutex is unlocked, the mutex is unlocked automatically by the destructor of the scoped_lock). At last the method notifies one of the consumer threads waiting for a new item to be inserted into the buffer.

The `pop_back()` method is called by the consumer thread in order to read the next item from the buffer. The method locks the mutex and waits until there is an unread item in the buffer. If there is at least one unread item, the method decrements the number of unread items and reads the next item from the `circular_buffer`. Then it unlocks the mutex and notifies one of the producer threads waiting for the buffer to free a space for the next item.

The `bounded buffer::pop_back()` method **does not remove the item** but the item is left in the circular_buffer which then **replaces it with a new one** (inserted by a producer) when the circular_buffer is full. This technique is more effective than removing the item explicitly by calling the `circular_buffer::pop_back()` method of the `circular_buffer`.

This claim is based on the assumption that an assignment (replacement) of a new item into an old one is more effective than a destruction (removal) of an old item and a consequent inplace construction (insertion) of a new item.

For comparison of bounded buffers based on different containers compile and run bounded_buffer_comparison.cpp. The test should reveal the bounded buffer based on the `circular_buffer` is most effective closely followed by the `std::deque` based bounded buffer. (In reality, the result may differ sometimes because the test is always affected by external factors such as immediate CPU load.)

You can see the full test code at bounded_buffer_comparison.cpp, and an example of output is

```
Description: Autorun "J:\Cpp\Misc\Debug\bounded_buffer_comparison.exe"
bounded_buffer<int> 5.15 s

bounded_buffer_space_optimized<int> 5.71 s

bounded_buffer_deque_based<int> 15.57 s

bounded_buffer_list_based<int> 17.33 s

bounded_buffer<std::string> 24.49 s

bounded_buffer_space_optimized<std::string> 28.33 s

bounded_buffer_deque_based<std::string> 29.45 s

bounded_buffer_list_based<std::string> 31.29 s
```

.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Header Files

The circular buffer library is defined in the file circular_buffer.hpp.

```
#include <boost/circular_buffer.hpp>
```

(There is also a forward declaration for the `circular_buffer` in the header file circular_buffer_fwd.hpp).

The `circular_buffer` is defined in the file base.hpp.

The `circular_buffer_space_optimized` is defined in the file space_optimized.hpp.

render X

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Modelled Concepts

Random Access Container, Front Insertion Sequence, and Back Insertion sequence

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Template Parameters

Table 1. Template parameter requirements

parameter	Requirements
T	The type of the elements stored in the circular_buffer. The T has to be Assignable and CopyConstructible. Moreover T has to be DefaultConstructible if supplied as a default parameter when invoking some of the circular_buffer's methods, e.g. `insert(iterator pos, const value_type& item = value_type())`. And EqualityComparable and/or LessThanComparable if the circular_buffer will be compared with another container.
Alloc	The allocator type used for all internal memory management. The Alloc has to meet the allocator requirements imposed by STL.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Trac Tickets

Report and view bugs and features by adding a ticket at Boost.Trac.

Existing open tickets for this library alone can be viewed here. Existing tickets for this library - including closed ones - can be viewed here.

Type: Bugs

#4100 Some boost classes have sizeof that depends on NDEBUG.

#5362 circular_buffer does not compile with BOOST_NO_EXCEPTIONS.

#6277 Checked iterators are not threadsafe.

#6747 Circular_Buffer / Bounded_Buffer inside Template class problem.

#7025 circular buffer reports warning: " type qualifiers ignored on function return type" while compile.

#7950 Eliminate W4-warnings under VS2005.

#8012 Inconsistency in `linearize()`.

#8438 `vector` & `circular_buffer` storage misbehave when using compiler optimizations.

Type: Feature Requests

#5511 Documentation needs some improvement.

#7888 circular_buffer should support move semantics.

Type: Patches

#8032 Warning fixes in circular_buffer.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Release Notes

Boost 1.56

- C++11 allocator model support implemented by Glen Fernandes using Boost allocator_traits.

Boost 1.55

- Documentation refactored by Paul A. Bristow using Quickbook, Doxygen and Autoindexing.
- Rvalue references emulation added by Antony Polukhin using Boost.Move.

Boost 1.42

- Added methods erase_begin(size_type) and erase_end(size_type) with constant complexity for such types of stored elements which do not need an explicit destruction e.g. int or double.
- Similarly changed implementation of the clear() method and the destructor so their complexity is now constant for such types of stored elements which do not require an explicit destruction (the complexity for other types remains linear).

Boost 1.37

```
*Added new methods is_linearized() and rotate(const_iterator).
```

- Fixed bugs: #1987 Patch to make circular_buffer.hpp #includes absolute. #1852 Copy constructor does not copy capacity.

Boost 1.36

- Changed behaviour of the circular_buffer(const allocator_type&) constructor. Since this version the constructor does not allocate any memory and both capacity and size are set to zero.
- Fixed bug: #1919 Default constructed circular buffer throws std::bad_alloc.

Boost 1.35

- Initial release.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Acknowledgements

Thomas Witt in 2002 produced a prototype called cyclic buffer.

The circular_buffer has a short history. Its first version was a std::deque adaptor. This container was not very effective because of many reallocations when inserting/removing an element. Thomas Wenish did a review of this version and motivated me to create a circular buffer which allocates memory at once when created.

The second version adapted `std::vector` but it has been abandoned soon because of limited control over iterator invalidation. The current version is a full-fledged STL compliant container.

Pavel Vozenilek did a thorough review of this version and came with many good ideas and improvements.

The idea of the space optimized circular buffer has been introduced by Pavel Vozenilek.

Also, I would like to thank Howard Hinnant, Nigel Stewart and everyone who participated at the formal review for valuable comments and ideas.

Paul A. Bristow refactored the documentation in 2013 to use the full power of Quickbook, Doxygen and Autoindexing.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Documentation Version Info

Last edit to Quickbook file circular_buffer.qbk was at 01:58:32 PM on 2014-Aug-14.

Tip

This should appear on the pdf version (but may be redundant on a html version where the last edit date is on the first (home) page).

Warning

Home page "Last revised" is GMT, not local time. Last edit date is local time.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Boost.Circular_buffer C++ Reference

Header <boost/circular_buffer.hpp>

```


BOOST_CB_ENABLE_DEBUG
BOOST_CB_ASSERT(Expr)
BOOST_CB_IS_CONVERTIBLE(Iterator, Type)
BOOST_CB_ASSERT_TEMPLATED_ITERATOR_CONSTRUCTORS
```

Macro BOOST_CB_ENABLE_DEBUG

BOOST_CB_ENABLE_DEBUG

Synopsis

```
// In header: <boost/circular_buffer.hpp>

BOOST_CB_ENABLE_DEBUG
```

Macro BOOST_CB_ASSERT

BOOST_CB_ASSERT

Synopsis

```
// In header: <boost/circular_buffer.hpp>

BOOST_CB_ASSERT(Expr)
```

Macro BOOST_CB_IS_CONVERTIBLE

BOOST_CB_IS_CONVERTIBLE

Synopsis

```
// In header: <boost/circular_buffer.hpp>

BOOST_CB_IS_CONVERTIBLE(Iterator, Type)
```

Macro BOOST_CB_ASSERT_TEMPLATED_ITERATOR_CONSTRUCTORS

BOOST_CB_ASSERT_TEMPLATED_ITERATOR_CONSTRUCTORS

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/circular_buffer.hpp>

BOOST_CB_ASSERT_TEMPLATED_ITERATOR_CONSTRUCTORS
```

Header <boost/circular_buffer/base.hpp>

```
namespace boost {
  template<typename T, typename Alloc> class circular_buffer;
  template<typename T, typename Alloc>
    bool operator==(const circular_buffer< T, Alloc > &,
                    const circular_buffer< T, Alloc > &);
  template<typename T, typename Alloc>
    bool operator<(const circular_buffer< T, Alloc > &,
                   const circular_buffer< T, Alloc > &);
  template<typename T, typename Alloc>
    bool operator!=(const circular_buffer< T, Alloc > &,
                    const circular_buffer< T, Alloc > &);
  template<typename T, typename Alloc>
    bool operator>(const circular_buffer< T, Alloc > &,
                   const circular_buffer< T, Alloc > &);
  template<typename T, typename Alloc>
    bool operator<=(const circular_buffer< T, Alloc > &,
                    const circular_buffer< T, Alloc > &);
  template<typename T, typename Alloc>
    bool operator>=(const circular_buffer< T, Alloc > &,
                    const circular_buffer< T, Alloc > &);
  template<typename T, typename Alloc>
    void swap(circular_buffer< T, Alloc > &, circular_buffer< T, Alloc > &);
}
```

Class template circular_buffer

boost::circular_buffer — Circular buffer - a STL compliant container.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/circular_buffer/base.hpp>

template<typename T, typename Alloc> 
class circular_buffer {
public:
  // types
  typedef circular_buffer< T, Alloc >                                                        ↵
                                           this_type;               // The type of this circu↵
lar_buffer.
  typedef boost::container::allocator_traits< Alloc >::value_type                            ↵
                                           value_type;              // The type of elements ↵
stored in the circular_buffer.
  typedef boost::container::allocator_traits< Alloc >::pointer                               ↵
                                          pointer;                 // A pointer to an element.
  typedef boost::container::allocator_traits< Alloc >::const_pointer                         ↵
                                          const_pointer;           // A const pointer to the ↵
element.
  typedef boost::container::allocator_traits< Alloc >::reference                             ↵
                                         reference;              // A reference to an element.
  typedef boost::container::allocator_traits< Alloc >::const_reference                       ↵
                                          const_reference;         // A const reference to an ↵
element.
  typedef boost::container::allocator_traits< Alloc >::difference_type                       ↵
                                           difference_type;
  typedef boost::container::allocator_traits< Alloc >::size_type                             ↵
                                           size_type;
  typedef Alloc                                                                              ↵
                                         allocator_type;          // The type of an allocator ↵
used in the circular_buffer.
  typedef cb_details::iterator< circular_buffer< T, Alloc >, cb_details::const_traits< boost::con↵
tainer::allocator_traits< Alloc > > >    const_iterator;          // A const (random access) ↵
iterator used to iterate through the circular_buffer.
  typedef cb_details::iterator< circular_buffer< T, Alloc >, cb_details::non↵
const_traits< boost::container::allocator_traits< Alloc > > > iterator;                // A (ran↵
dom access) iterator used to iterate through the circular_buffer.
  typedef boost::reverse_iterator< const_iterator >                                          ↵
                                         const_reverse_iterator;  // A const iterator used to ↵
iterate backwards through a circular_buffer.
  typedef boost::reverse_iterator< iterator >                                                ↵
                                          reverse_iterator;        // An iterator used to iter↵
ate backwards through a circular_buffer.
  typedef std::pair< pointer, size_type >                                                    ↵
                                           array_range;
  typedef std::pair< const_pointer, size_type >                                              ↵
                                           const_array_range;
  typedef size_type                                                                          ↵
                                           capacity_type;
  typedef const value_type &                                                                 ↵
                                          param_value_type;        // A type representing the ↵
"best" way to pass the value_type to a method.
  typedef value_type &&                                                                      ↵
                                           rvalue_type;

  // construct/copy/destruct
  explicit circular_buffer(const allocator_type & = allocator_type()) noexcept;
  explicit circular_buffer(capacity_type, 
                           const allocator_type & = allocator_type());
  circular_buffer(size_type, param_value_type, 
                  const allocator_type & = allocator_type());
  circular_buffer(capacity_type, size_type, param_value_type,
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
                const allocator_type & = allocator_type());
circular_buffer(const circular_buffer< T, Alloc > &);
circular_buffer(circular_buffer< T, Alloc > &&) noexcept;
template<typename InputIterator>
  circular_buffer(InputIterator, InputIterator,
                  const allocator_type & = allocator_type());
template<typename InputIterator>
  circular_buffer(capacity_type, InputIterator, InputIterator,
                  const allocator_type & = allocator_type());
circular_buffer< T, Alloc > & operator=(const circular_buffer< T, Alloc > &);
circular_buffer< T, Alloc > &
operator=(circular_buffer< T, Alloc > &&) noexcept;
~circular_buffer();

// public member functions
allocator_type get_allocator() const noexcept;
allocator_type & get_allocator() noexcept;
iterator begin() noexcept;
iterator end() noexcept;
const_iterator begin() const noexcept;
const_iterator end() const noexcept;
reverse_iterator rbegin() noexcept;
reverse_iterator rend() noexcept;
const_reverse_iterator rbegin() const noexcept;
const_reverse_iterator rend() const noexcept;
reference operator[](size_type);
const_reference operator[](size_type) const;
reference at(size_type);
const_reference at(size_type) const;
reference front();
reference back();
const_reference front() const;
const_reference back() const;
array_range array_one();
array_range array_two();
const_array_range array_one() const;
const_array_range array_two() const;
pointer linearize();
bool is_linearized() const noexcept;
void rotate(const_iterator);
size_type size() const noexcept;
size_type max_size() const noexcept;
bool empty() const noexcept;
bool full() const noexcept;
size_type reserve() const noexcept;
capacity_type capacity() const noexcept;
void set_capacity(capacity_type);
void resize(size_type, param_value_type = value_type());
void rset_capacity(capacity_type);
void rresize(size_type, param_value_type = value_type());
void assign(size_type, param_value_type);
void assign(capacity_type, size_type, param_value_type);
template<typename InputIterator> void assign(InputIterator, InputIterator);
template<typename InputIterator>
  void assign(capacity_type, InputIterator, InputIterator);
void swap(circular_buffer< T, Alloc > &) noexcept;
void push_back(param_value_type);
void push_back(rvalue_type);
void push_back();
void push_front(param_value_type);
void push_front(rvalue_type);
void push_front();
void pop_back();
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
  void pop_front();
  iterator insert(iterator, param_value_type);
  iterator insert(iterator, rvalue_type);
  iterator insert(iterator);
  void insert(iterator, size_type, param_value_type);
  template<typename InputIterator>
    void insert(iterator, InputIterator, InputIterator);
  iterator rinsert(iterator, param_value_type);
  iterator rinsert(iterator, rvalue_type);
  iterator rinsert(iterator);
  void rinsert(iterator, size_type, param_value_type);
  template<typename InputIterator>
    void rinsert(iterator, InputIterator, InputIterator);
  iterator erase(iterator);
  iterator erase(iterator, iterator);
  iterator rerase(iterator);
  iterator rerase(iterator, iterator);
  void erase_begin(size_type);
  void erase_end(size_type);
  void clear() noexcept;

  // private member functions
  template<typename ValT> void push_back_impl(ValT);
  template<typename ValT> void push_front_impl(ValT);
  template<typename ValT> iterator insert_impl(iterator, ValT);
  template<typename ValT> iterator rinsert_impl(iterator, ValT);
  void check_position(size_type) const;
  template<typename Pointer> void increment(Pointer &) const;
  template<typename Pointer> void decrement(Pointer &) const;
  template<typename Pointer> Pointer add(Pointer, difference_type) const;
  template<typename Pointer> Pointer sub(Pointer, difference_type) const;
  pointer map_pointer(pointer) const;
  pointer allocate(size_type);
  void deallocate(pointer, size_type);
  bool is_uninitialized(const_pointer) const noexcept;
  void replace(pointer, param_value_type);
  void replace(pointer, rvalue_type);
  void construct_or_replace(bool, pointer, param_value_type);
  void construct_or_replace(bool, pointer, rvalue_type);
  void destroy_item(pointer);
  void destroy_if_constructed(pointer);
  void destroy_content();
  void destroy_content(const true_type &);
  void destroy_content(const false_type &);
  void destroy() noexcept;
  void initialize_buffer(capacity_type);
  void initialize_buffer(capacity_type, param_value_type);
  template<typename IntegralType>
    void initialize(IntegralType, IntegralType, const true_type &);
  template<typename Iterator>
    void initialize(Iterator, Iterator, const false_type &);
  template<typename InputIterator>
    void initialize(InputIterator, InputIterator,
                    const std::input_iterator_tag &);
  template<typename ForwardIterator>
    void initialize(ForwardIterator, ForwardIterator,
                    const std::forward_iterator_tag &);
  template<typename IntegralType>
    void initialize(capacity_type, IntegralType, IntegralType,
                    const true_type &);
  template<typename Iterator>
    void initialize(capacity_type, Iterator, Iterator, const false_type &);
  template<typename InputIterator>
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
    void initialize(capacity_type, InputIterator, InputIterator,
                    const std::input_iterator_tag &);
  template<typename ForwardIterator>
    void initialize(capacity_type, ForwardIterator, ForwardIterator,
                    const std::forward_iterator_tag &);
  template<typename ForwardIterator>
    void initialize(capacity_type, ForwardIterator, ForwardIterator,
                    size_type);
  void reset(pointer, pointer, capacity_type);
  void swap_allocator(circular_buffer< T, Alloc > &, const true_type &);
  void swap_allocator(circular_buffer< T, Alloc > &, const false_type &);
  template<typename IntegralType>
    void assign(IntegralType, IntegralType, const true_type &);
  template<typename Iterator>
    void assign(Iterator, Iterator, const false_type &);
  template<typename InputIterator>
    void assign(InputIterator, InputIterator, const std::input_iterator_tag &);
  template<typename ForwardIterator>
    void assign(ForwardIterator, ForwardIterator,
                const std::forward_iterator_tag &);
  template<typename IntegralType>
    void assign(capacity_type, IntegralType, IntegralType, const true_type &);
  template<typename Iterator>
    void assign(capacity_type, Iterator, Iterator, const false_type &);
  template<typename InputIterator>
    void assign(capacity_type, InputIterator, InputIterator,
                const std::input_iterator_tag &);
  template<typename ForwardIterator>
    void assign(capacity_type, ForwardIterator, ForwardIterator,
                const std::forward_iterator_tag &);
  template<typename Functor>
    void assign_n(capacity_type, size_type, const Functor &);
  template<typename ValT> iterator insert_item(const iterator &, ValT);
  template<typename IntegralType>
    void insert(const iterator &, IntegralType, IntegralType,
                const true_type &);
  template<typename Iterator>
    void insert(const iterator &, Iterator, Iterator, const false_type &);
  template<typename InputIterator>
    void insert(iterator, InputIterator, InputIterator,
                const std::input_iterator_tag &);
  template<typename ForwardIterator>
    void insert(const iterator &, ForwardIterator, ForwardIterator,
                const std::forward_iterator_tag &);
  template<typename Wrapper>
    void insert_n(const iterator &, size_type, const Wrapper &);
  template<typename IntegralType>
    void rinsert(const iterator &, IntegralType, IntegralType,
                 const true_type &);
  template<typename Iterator>
    void rinsert(const iterator &, Iterator, Iterator, const false_type &);
  template<typename InputIterator>
    void rinsert(iterator, InputIterator, InputIterator,
                 const std::input_iterator_tag &);
  template<typename ForwardIterator>
    void rinsert(const iterator &, ForwardIterator, ForwardIterator,
                 const std::forward_iterator_tag &);
  template<typename Wrapper>
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
    void rinsert_n(const iterator &, size_type, const Wrapper &);
  void erase_begin(size_type, const true_type &);
  void erase_begin(size_type, const false_type &);
  void erase_end(size_type, const true_type &);
  void erase_end(size_type, const false_type &);
};
```

Description

Type Requirements T. The `T` has to be SGIAssignable (SGI STL defined combination of Assignable and CopyConstructible). Moreover `T` has to be DefaultConstructible if supplied as a default parameter when invoking some of the `circular_buffer`'s methods e.g. `insert(iterator pos, const value_type& item = value_type())`. And EqualityComparable and/or LessThanComparable if the `circular_buffer` will be compared with another container.

Type Requirements Alloc. The `Alloc` has to meet the allocator requirements imposed by STL.

Default Alloc. std::allocator<T>
For detailed documentation of the circular_buffer visit: http://www.boost.org/libs/circular_buffer/doc/circular_buffer.html

Template Parameters

1.
   ```
   typename T
   ```

   The type of the elements stored in the `circular_buffer`.

2.
   ```
   typename Alloc
   ```

   The allocator type used for all internal memory management.

`circular_buffer` public types

1. typedef boost::container::allocator_traits< Alloc >::difference_type difference_type;

   (A signed integral type used to represent the distance between two iterators.)

2. typedef boost::container::allocator_traits< Alloc >::size_type size_type;

   (An unsigned integral type that can represent any non-negative value of the container's distance type.)

3. typedef std::pair< pointer, size_type > array_range;

   (A typedef for the `std::pair` where its first element is a pointer to a beginning of an array and its second element represents a size of the array.)

4. typedef std::pair< const_pointer, size_type > const_array_range;

   (A typedef for the `std::pair` where its first element is a pointer to a beginning of a const array and its second element represents a size of the const array.)

5. typedef size_type capacity_type;

   (Same as `size_type` - defined for consistency with the __cbso class.

6. typedef value_type && rvalue_type;

   A type representing rvalue from param type. On compilers without rvalue references support this type is the Boost.Moves type used for emulation.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

`circular_buffer` public construct/copy/destruct

1.

```
explicit circular_buffer(const allocator_type & alloc = allocator_type()) noexcept;
```

Create an empty `circular_buffer` with zero capacity.

Complexity. Constant.

> **Warning**
>
> Since Boost version 1.36 the behaviour of this constructor has changed. Now the constructor does not allocate any memory and both capacity and size are set to zero. Also note when inserting an element into a `circular_buffer` with zero capacity (e.g. by `push_back(const_reference)` or `insert(iterator, value_type)`) nothing will be inserted and the size (as well as capacity) remains zero.

> **Note**
>
> You can explicitly set the capacity by calling the `set_capacity(capacity_type)` method or you can use the other constructor with the capacity specified.

See Also:

`circular_buffer(capacity_type, const allocator_type& alloc)`, `set_capacity(capacity_type)`

Parameters:	`alloc`	The allocator.
Postconditions:	`capacity() == 0 && size() == 0`	
Throws:	Nothing.	

2.

```
explicit circular_buffer(capacity_type buffer_capacity,
                         const allocator_type & alloc = allocator_type());
```

Create an empty `circular_buffer` with the specified capacity.

Complexity. Constant.

Parameters:	`alloc`	The allocator.
	`buffer_capacity`	The maximum number of elements which can be stored in the `circular_buffer`.
Postconditions:	`capacity() == buffer_capacity && size() == 0`	
Throws:	An allocation error if memory is exhausted (`std::bad_alloc` if the standard allocator is used).	

3.

```
circular_buffer(size_type n, param_value_type item,
                const allocator_type & alloc = allocator_type());
```

Create a full `circular_buffer` with the specified capacity and filled with `n` copies of `item`.

Complexity. Linear (in the `n`).

Parameters:	`alloc`	The allocator.
	`item`	The element the created `circular_buffer` will be filled with.
	`n`	The number of elements the created `circular_buffer` will be filled with.
Postconditions:	`capacity() == n && full() && (*this)[0] == item && (*this)[1] == item && ... && (*this)[n - 1] == item`	
Throws:	An allocation error if memory is exhausted (`std::bad_alloc` if the standard allocator is used). Whatever `T::T(const T&)` throws.	

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

4.

```
circular_buffer(capacity_type buffer_capacity, size_type n,
                param_value_type item,
                const allocator_type & alloc = allocator_type());
```

Create a `circular_buffer` with the specified capacity and filled with `n` copies of `item`.

Complexity. Linear (in the `n`).

Parameters:

`alloc`	The allocator.
`buffer_capacity`	The capacity of the created `circular_buffer`.
`item`	The element the created `circular_buffer` will be filled with.
`n`	The number of elements the created `circular_buffer` will be filled with.

Requires: `buffer_capacity >= n`

Postconditions: `capacity() == buffer_capacity && size() == n && (*this)[0] == item && (*this)[1] == item && ... && (*this)[n - 1] == item`

Throws: An allocation error if memory is exhausted (`std::bad_alloc` if the standard allocator is used). Whatever `T::T(const T&)` throws.

5.

```
circular_buffer(const circular_buffer< T, Alloc > & cb);
```

The copy constructor.

Creates a copy of the specified `circular_buffer`.

Complexity. Linear (in the size of `cb`).

Parameters:

`cb`	The `circular_buffer` to be copied.

Postconditions: `*this == cb`

Throws: An allocation error if memory is exhausted (`std::bad_alloc` if the standard allocator is used). Whatever `T::T(const T&)` throws.

6.

```
circular_buffer(circular_buffer< T, Alloc > && cb) noexcept;
```

The move constructor.

Move constructs a `circular_buffer` from `cb`, leaving `cb` empty.

Constant.

Parameters:

`cb`	`circular_buffer` to 'steal' value from.

Requires: C++ compiler with rvalue references support.

Postconditions: `cb.empty()`

Throws: Nothing.

7.

```
template<typename InputIterator>
  circular_buffer(InputIterator first, InputIterator last,
                  const allocator_type & alloc = allocator_type());
```

Create a full `circular_buffer` filled with a copy of the range.

Complexity. Linear (in the `std::distance(first, last)`).

Parameters:

`alloc`	The allocator.
`first`	The beginning of the range to be copied.
`last`	The end of the range to be copied.

Requires: Valid range `[first, last)`.
`first` and `last` have to meet the requirements of InputIterator.

Postconditions: `capacity() == std::distance(first, last) && full() && (*this)[0]== *first && (*this)[1] == *(first + 1) && ... && (*this)[std::distance(first, last) - 1] == *(last - 1)`

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Throws:	An allocation error if memory is exhausted (`std::bad_alloc` if the standard allocator is used). Whatever `T::T(const T&)` throws.

8.

```
template<typename InputIterator>
  circular_buffer(capacity_type buffer_capacity, InputIterator first,
                  InputIterator last,
                  const allocator_type & alloc = allocator_type());
```

Create a `circular_buffer` with the specified capacity and filled with a copy of the range.

Complexity. Linear (in `std::distance(first, last)`; in `min[capacity, std::distance(first, last)]` if the `InputIterator` is a RandomAccessIterator).

Parameters:	`alloc`	The allocator.
	`buffer_capacity`	The capacity of the created `circular_buffer`.
	`first`	The beginning of the range to be copied.
	`last`	The end of the range to be copied.
Requires:	Valid range `[first, last)`. `first` and `last` have to meet the requirements of InputIterator.	
Postconditions:	`capacity() == buffer_capacity && size() <= std::distance(first, last) && (*this)[0]== *(last - buffer_capacity) && (*this)[1] == *(last - buffer_capacity + 1) && ... && (*this)[buffer_capacity - 1] == *(last - 1)` If the number of items to be copied from the range `[first, last)` is greater than the specified `buffer_capacity` then only elements from the range `[last - buffer_capacity, last)` will be copied.	
Throws:	An allocation error if memory is exhausted (`std::bad_alloc` if the standard allocator is used). Whatever `T::T(const T&)` throws.	

9.

```
circular_buffer< T, Alloc > &
operator=(const circular_buffer< T, Alloc > & cb);
```

The assign operator.

Makes this `circular_buffer` to become a copy of the specified `circular_buffer`.

Exception Safety. Strong.

Iterator Invalidation. Invalidates all iterators pointing to this `circular_buffer` (except iterators equal to `end()`).

Complexity. Linear (in the size of `cb`).

See Also:

`assign(size_type, const_reference)`, `assign(capacity_type, size_type, const_reference)`, `assign(InputIterator, InputIterator)`, `assign(capacity_type, InputIterator, InputIterator)`

Parameters:	`cb`	The `circular_buffer` to be copied.
Postconditions:	`*this == cb`	
Throws:	An allocation error if memory is exhausted (`std::bad_alloc` if the standard allocator is used). Whatever `T::T(const T&)` throws.	

10.

```
circular_buffer< T, Alloc > &
operator=(circular_buffer< T, Alloc > && cb) noexcept;
```

Move assigns content of `cb` to `*this`, leaving `cb` empty.

Complexity. Constant.

Parameters:	`cb`	`circular_buffer` to 'steal' value from.
Requires:	C++ compiler with rvalue references support.	
Postconditions:	`cb.empty()`	

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Throws: Nothing.

11.

```
~circular_buffer();
```

The destructor.

Destroys the `circular_buffer`.

Iterator Invalidation. Invalidates all iterators pointing to the `circular_buffer` (including iterators equal to `end()`).

Complexity. Constant (in the size of the `circular_buffer`) for scalar types; linear for other types.

See Also:

`clear()`
Throws: Nothing.

`circular_buffer` public member functions

1.

```
allocator_type get_allocator() const noexcept;
```

Get the allocator.

Exception Safety. No-throw.

Iterator Invalidation. Does not invalidate any iterators.

Complexity. Constant (in the size of the `circular_buffer`).

See Also:

`get_allocator()` for obtaining an allocator reference.
Returns: The allocator.
Throws: Nothing.

2.

```
allocator_type & get_allocator() noexcept;
```

Get the allocator reference.

Exception Safety. No-throw.

Iterator Invalidation. Does not invalidate any iterators.

Complexity. Constant (in the size of the `circular_buffer`).

> **Note**
>
> This method was added in order to optimize obtaining of the allocator with a state, although use of stateful allocators in STL is discouraged.

See Also:

`get_allocator() const`
Returns: A reference to the allocator.
Throws: Nothing.

3.

```
iterator begin() noexcept;
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Get the iterator pointing to the beginning of the `circular_buffer`.

Exception Safety. No-throw.

Iterator Invalidation. Does not invalidate any iterators.

Complexity. Constant (in the size of the `circular_buffer`).

See Also:

`end()`, `rbegin()`, `rend()`

Returns:	A random access iterator pointing to the first element of the `circular_buffer`. If the `circular_buffer` is empty it returns an iterator equal to the one returned by `end()`.
Throws:	Nothing.

4.
```
iterator end() noexcept;
```

Get the iterator pointing to the end of the `circular_buffer`.

Exception Safety. No-throw.

Iterator Invalidation. Does not invalidate any iterators.

Complexity. Constant (in the size of the `circular_buffer`).

See Also:

`begin()`, `rbegin()`, `rend()`

Returns:	A random access iterator pointing to the element "one behind" the last element of the `circular_buffer`. If the `circular_buffer` is empty it returns an iterator equal to the one returned by `begin()`.
Throws:	Nothing.

5.
```
const_iterator begin() const noexcept;
```

Get the const iterator pointing to the beginning of the `circular_buffer`.

Exception Safety. No-throw.

Iterator Invalidation. Does not invalidate any iterators.

Complexity. Constant (in the size of the `circular_buffer`).

See Also:

`end() const`, `rbegin() const`, `rend() const`

Returns:	A const random access iterator pointing to the first element of the `circular_buffer`. If the `circular_buffer` is empty it returns an iterator equal to the one returned by `end() const`.
Throws:	Nothing.

6.
```
const_iterator end() const noexcept;
```

Get the const iterator pointing to the end of the `circular_buffer`.

Exception Safety. No-throw.

Iterator Invalidation. Does not invalidate any iterators.

Complexity. Constant (in the size of the `circular_buffer`).

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

See Also:

`begin() const`, `rbegin() const`, `rend() const`

Returns: A const random access iterator pointing to the element "one behind" the last element of the `circular_buffer`. If the `circular_buffer` is empty it returns an iterator equal to the one returned by `begin() const` const.

Throws: Nothing.

7.
```
reverse_iterator rbegin() noexcept;
```

Get the iterator pointing to the beginning of the "reversed" `circular_buffer`.

Exception Safety. No-throw.

Iterator Invalidation. Does not invalidate any iterators.

Complexity. Constant (in the size of the `circular_buffer`).

See Also:

`rend()`, `begin()`, `end()`

Returns: A reverse random access iterator pointing to the last element of the `circular_buffer`. If the `circular_buffer` is empty it returns an iterator equal to the one returned by `rend()`.

Throws: Nothing.

8.
```
reverse_iterator rend() noexcept;
```

Get the iterator pointing to the end of the "reversed" `circular_buffer`.

Exception Safety. No-throw.

Iterator Invalidation. Does not invalidate any iterators.

Complexity. Constant (in the size of the `circular_buffer`).

See Also:

`rbegin()`, `begin()`, `end()`

Returns: A reverse random access iterator pointing to the element "one before" the first element of the `circular_buffer`. If the `circular_buffer` is empty it returns an iterator equal to the one returned by `rbegin()`.

Throws: Nothing.

9.
```
const_reverse_iterator rbegin() const noexcept;
```

Get the const iterator pointing to the beginning of the "reversed" `circular_buffer`.

Exception Safety. No-throw.

Iterator Invalidation. Does not invalidate any iterators.

Complexity. Constant (in the size of the `circular_buffer`).

See Also:

`rend() const`, `begin() const`, `end() const`

Returns: A const reverse random access iterator pointing to the last element of the `circular_buffer`. If the `circular_buffer` is empty it returns an iterator equal to the one returned by `rend() const`.

Throws: Nothing.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

10.
```
const_reverse_iterator rend() const noexcept;
```

Get the const iterator pointing to the end of the "reversed" `circular_buffer`.

Exception Safety. No-throw.

Iterator Invalidation. Does not invalidate any iterators.

Complexity. Constant (in the size of the `circular_buffer`).

See Also:

`rbegin() const`, `begin() const`, `end() const`

Returns:	A const reverse random access iterator pointing to the element "one before" the first element of the `circular_buffer`. If the `circular_buffer` is empty it returns an iterator equal to the one returned by `rbegin() const`.
Throws:	Nothing.

11.
```
reference operator[](size_type index);
```

Get the element at the `index` position.

Exception Safety. No-throw.

Iterator Invalidation. Does not invalidate any iterators.

Complexity. Constant (in the size of the `circular_buffer`).

See Also:

`at()`

Parameters:	`index` The position of the element.
Requires:	`0 <= index && index < size()`
Returns:	A reference to the element at the `index` position.
Throws:	Nothing.

12.
```
const_reference operator[](size_type index) const;
```

Get the element at the `index` position.

Exception Safety. No-throw.

Iterator Invalidation. Does not invalidate any iterators.

Complexity. Constant (in the size of the `circular_buffer`).

See Also:

`at() const`

Parameters:	`index` The position of the element.
Requires:	`0 <= index && index < size()`
Returns:	A const reference to the element at the `index` position.
Throws:	Nothing.

13.
```
reference at(size_type index);
```

Get the element at the `index` position.

Exception Safety. Strong.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Iterator Invalidation. Does not invalidate any iterators.

Complexity. Constant (in the size of the `circular_buffer`).

See Also:

`operator[]`

Parameters:	`index` The position of the element.
Returns:	A reference to the element at the `index` position.
Throws:	<code>std::out_of_range</code> when the `index` is invalid (when `index >= size()`).

14.
```
const_reference at(size_type index) const;
```

Get the element at the `index` position.

Exception Safety. Strong.

Iterator Invalidation. Does not invalidate any iterators.

Complexity. Constant (in the size of the `circular_buffer`).

See Also:

`operator[] const`

Parameters:	`index` The position of the element.
Returns:	A const reference to the element at the `index` position.
Throws:	<code>std::out_of_range</code> when the `index` is invalid (when `index >= size()`).

15.
```
reference front();
```

Get the first element.

Exception Safety. No-throw.

Iterator Invalidation. Does not invalidate any iterators.

Complexity. Constant (in the size of the `circular_buffer`).

See Also:

`back()`

Requires:	`!empty()`
Returns:	A reference to the first element of the `circular_buffer`.
Throws:	Nothing.

16.
```
reference back();
```

Get the last element.

Exception Safety. No-throw.

Iterator Invalidation. Does not invalidate any iterators.

Complexity. Constant (in the size of the `circular_buffer`).

See Also:

`front()`

Requires:	`!empty()`

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Returns: A reference to the last element of the `circular_buffer`.
Throws: Nothing.

17.
```
const_reference front() const;
```

Get the first element.

Exception Safety. No-throw.

Iterator Invalidation. Does not invalidate any iterators.

Complexity. Constant (in the size of the `circular_buffer`).

See Also:

`back() const`
Requires: `!empty()`
Returns: A const reference to the first element of the `circular_buffer`.
Throws: Nothing.

18.
```
const_reference back() const;
```

Get the last element.

Exception Safety. No-throw.

Iterator Invalidation. Does not invalidate any iterators.

Complexity. Constant (in the size of the `circular_buffer`).

See Also:

`front() const`
Requires: `!empty()`
Returns: A const reference to the last element of the `circular_buffer`.
Throws: Nothing.

19.
```
array_range array_one();
```

Get the first continuous array of the internal buffer.

This method in combination with `array_two()` can be useful when passing the stored data into a legacy C API as an array. Suppose there is a `circular_buffer` of capacity 10, containing 7 characters `'a', 'b', ..., 'g'` where `buff[0] == 'a'`, `buff[1] == 'b'`, ... and `buff[6] == 'g'`:
```
circular_buffer<char> buff(10);
```
The internal representation is often not linear and the state of the internal buffer may look like this:
```
|e|f|g| | | |a|b|c|d|
end ___^
begin _______^
```

where `|a|b|c|d|` represents the "array one", `|e|f|g|` represents the "array two" and `| | | |` is a free space.
Now consider a typical C style function for writing data into a file:
```
int write(int file_desc, char* buff, int num_bytes);
```
There are two ways how to write the content of the `circular_buffer` into a file. Either relying on `array_one()` and `array_two()` methods and calling the write function twice:
```
array_range ar = buff.array_one();
write(file_desc, ar.first, ar.second);
ar = buff.array_two();
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
write(file_desc, ar.first, ar.second);
```

Or relying on the `linearize()` method:

```
write(file_desc, buff.linearize(), buff.size());
```

Since the complexity of `array_one()` and `array_two()` methods is constant the first option is suitable when calling the write method is "cheap". On the other hand the second option is more suitable when calling the write method is more "expensive" than calling the `linearize()` method whose complexity is linear.

Exception Safety. No-throw.

Iterator Invalidation. Does not invalidate any iterators.

Complexity. Constant (in the size of the `circular_buffer`).

Warning

In general invoking any method which modifies the internal state of the `circular_buffer` may delinearize the internal buffer and invalidate the array ranges returned by `array_one()` and `array_two()` (and their const versions).

Note

In the case the internal buffer is linear e.g. `|a|b|c|d|e|f|g| | | |` the "array one" is represented by `|a|b|c|d|e|f|g|` and the "array two" does not exist (the `array_two()` method returns an array with the size `0`).

See Also:

`array_two()`, `linearize()`

Returns:	The array range of the first continuous array of the internal buffer. In the case the `circular_buffer` is empty the size of the returned array is `0`.
Throws:	Nothing.

20.
```
array_range array_two();
```

Get the second continuous array of the internal buffer.

This method in combination with `array_one()` can be useful when passing the stored data into a legacy C API as an array.

Exception Safety. No-throw.

Iterator Invalidation. Does not invalidate any iterators.

Complexity. Constant (in the size of the `circular_buffer`).

See Also:

`array_one()`

Returns:	The array range of the second continuous array of the internal buffer. In the case the internal buffer is linear or the `circular_buffer` is empty the size of the returned array is `0`.
Throws:	Nothing.

21.
```
const_array_range array_one() const;
```

Get the first continuous array of the internal buffer.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

This method in combination with `array_two() const` can be useful when passing the stored data into a legacy C API as an array.

Exception Safety. No-throw.

Iterator Invalidation. Does not invalidate any iterators.

Complexity. Constant (in the size of the `circular_buffer`).

See Also:

`array_two() const`; `array_one()` for more details how to pass data into a legacy C API.
Returns: The array range of the first continuous array of the internal buffer. In the case the `circular_buffer` is empty the size of the returned array is `0`.
Throws: Nothing.

22.
```
const_array_range array_two() const;
```

Get the second continuous array of the internal buffer.

This method in combination with `array_one() const` can be useful when passing the stored data into a legacy C API as an array.

Exception Safety. No-throw.

Iterator Invalidation. Does not invalidate any iterators.

Complexity. Constant (in the size of the `circular_buffer`).

See Also:

`array_one() const`
Returns: The array range of the second continuous array of the internal buffer. In the case the internal buffer is linear or the `circular_buffer` is empty the size of the returned array is `0`.
Throws: Nothing.

23.
```
pointer linearize();
```

Linearize the internal buffer into a continuous array.

This method can be useful when passing the stored data into a legacy C API as an array.

Exception Safety. Basic; no-throw if the operations in the *Throws* section do not throw anything.

Iterator Invalidation. Invalidates all iterators pointing to the `circular_buffer` (except iterators equal to `end()`); does not invalidate any iterators if the postcondition (the *Effect*) is already met prior calling this method.

Complexity. Linear (in the size of the `circular_buffer`); constant if the postcondition (the *Effect*) is already met.

Warning

In general invoking any method which modifies the internal state of the `circular_buffer` may delinearize the internal buffer and invalidate the returned pointer.

See Also:

`array_one()` and `array_two()` for the other option how to pass data into a legacy C API; `is_linearized()`, `rotate(const_iterator)`

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Postconditions:	`&(*this)[0] < &(*this)[1] < ... < &(*this)[size() - 1]`
Returns:	A pointer to the beginning of the array or `0` if empty.
Throws:	<a href="circular_buffer/implementation.html#circular_buffer.implementation.exceptions_of_move_if_noexcept_t">Exceptions of move_if_noexcept(T&).

24.

```
bool is_linearized() const noexcept;
```

Is the `circular_buffer` linearized?

Exception Safety. No-throw.

Iterator Invalidation. Does not invalidate any iterators.

Complexity. Constant (in the size of the `circular_buffer`).

See Also:

`linearize()`, `array_one()`, `array_two()`

Returns:	`true` if the internal buffer is linearized into a continuous array (i.e. the `circular_buffer` meets a condition `&(*this)[0] < &(*this)[1] < ... < &(*this)[size() - 1]`); `false` otherwise.
Throws:	Nothing.

25.

```
void rotate(const_iterator new_begin);
```

Rotate elements in the `circular_buffer`.

A more effective implementation of `std::rotate`.

Exception Safety. Basic; no-throw if the `circular_buffer` is full or `new_begin` points to `begin()` or if the operations in the *Throws* section do not throw anything.

Iterator Invalidation. If `m < n` invalidates iterators pointing to the last `m` elements (**including** `new_begin`, but not iterators equal to `end()`) else invalidates iterators pointing to the first `n` elements; does not invalidate any iterators if the `circular_buffer` is full.

Complexity. Linear (in `(std::min)(m, n)`); constant if the `circular_buffer` is full.

See Also:

`std::rotate`

Parameters:	`new_begin` The new beginning.
Requires:	`new_begin` is a valid iterator pointing to the `circular_buffer` **except** its end.

Postconditions:	Before calling the method suppose:
	`m == std::distance(new_begin, end())`
	`n == std::distance(begin(), new_begin)`
	`val_0 == *new_begin, val_1 == *(new_begin + 1), ... val_m == *(new_begin + m)`
	`val_r1 == *(new_begin - 1), val_r2 == *(new_begin - 2), ... val_rn == *(new_begin - n)`
	then after call to the method:
	`val_0 == (*this)[0] && val_1 == (*this)[1] && ... && val_m == (*this)[m - 1] && val_r1 == (*this)[m + n - 1] && val_r2 == (*this)[m + n - 2] && ... && val_rn == (*this)[m]`

Throws:	See Exceptions of move_if_noexcept(T&).

26.

```
size_type size() const noexcept;
```

Get the number of elements currently stored in the `circular_buffer`.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Exception Safety. No-throw.

Iterator Invalidation. Does not invalidate any iterators.

Complexity. Constant (in the size of the `circular_buffer`).

See Also:

`capacity()`, `max_size()`, `reserve()`, `resize(size_type, const_reference)`
Returns: The number of elements stored in the `circular_buffer`.
Throws: Nothing.

27.
```
size_type max_size() const noexcept;
```

Get the largest possible size or capacity of the `circular_buffer`. (It depends on allocator's max_size()).

Exception Safety. No-throw.

Iterator Invalidation. Does not invalidate any iterators.

Complexity. Constant (in the size of the `circular_buffer`).

See Also:

`size()`, `capacity()`, `reserve()`
Returns: The maximum size/capacity the `circular_buffer` can be set to.
Throws: Nothing.

28.
```
bool empty() const noexcept;
```

Is the `circular_buffer` empty?

Exception Safety. No-throw.

Iterator Invalidation. Does not invalidate any iterators.

Complexity. Constant (in the size of the `circular_buffer`).

See Also:

`full()`
Returns: `true` if there are no elements stored in the `circular_buffer`; `false` otherwise.
Throws: Nothing.

29.
```
bool full() const noexcept;
```

Is the `circular_buffer` full?

Exception Safety. No-throw.

Iterator Invalidation. Does not invalidate any iterators.

Complexity. Constant (in the size of the `circular_buffer`).

See Also:

`empty()`
Returns: `true` if the number of elements stored in the `circular_buffer` equals the capacity of the `circular_buffer`; `false` otherwise.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Throws: Nothing.

30.
```
size_type reserve() const noexcept;
```

Get the maximum number of elements which can be inserted into the `circular_buffer` without overwriting any of already stored elements.

Exception Safety. No-throw.

Iterator Invalidation. Does not invalidate any iterators.

Complexity. Constant (in the size of the `circular_buffer`).

See Also:

`capacity()`, `size()`, `max_size()`

Returns:	`capacity() - size()`
Throws:	Nothing.

31.
```
capacity_type capacity() const noexcept;
```

Get the capacity of the `circular_buffer`.

Exception Safety. No-throw.

Iterator Invalidation. Does not invalidate any iterators.

Complexity. Constant (in the size of the `circular_buffer`).

See Also:

`reserve()`, `size()`, `max_size()`, `set_capacity(capacity_type)`

Returns:	The maximum number of elements which can be stored in the `circular_buffer`.
Throws:	Nothing.

32.
```
void set_capacity(capacity_type new_capacity);
```

Change the capacity of the `circular_buffer`.

Exception Safety. Strong.

Iterator Invalidation. Invalidates all iterators pointing to the `circular_buffer` (except iterators equal to `end()`) if the new capacity is different from the original.

Complexity. Linear (in `min[size(), new_capacity]`).

See Also:

`rset_capacity(capacity_type)`, `resize(size_type, const_reference)`

Parameters:	`new_capacity` The new capacity.
Requires:	If `T` is a move only type, then compiler shall support `noexcept` modifiers and move constructor of `T` must be marked with it (must not throw exceptions).
Postconditions:	`capacity() == new_capacity && size() <= new_capacity` If the current number of elements stored in the `circular_buffer` is greater than the desired new capacity then number of `[size() - new_capacity]` **last** elements will be removed and the new size will be equal to `new_capacity`.
Throws:	An allocation error if memory is exhausted, (`std::bad_alloc` if the standard allocator is used). Whatever `T::T(const T&)` throws or nothing if `T::T(T&&)` is noexcept.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

33.
```
void resize(size_type new_size, param_value_type item = value_type());
```

Change the size of the `circular_buffer`.

Exception Safety. Basic.

Iterator Invalidation. Invalidates all iterators pointing to the `circular_buffer` (except iterators equal to `end()`) if the new size is greater than the current capacity. Invalidates iterators pointing to the removed elements if the new size is lower that the original size. Otherwise it does not invalidate any iterator.

Complexity. Linear (in the new size of the `circular_buffer`).

See Also:

`rresize(size_type, const_reference)`, `set_capacity(capacity_type)`

Parameters:	`item`	The element the `circular_buffer` will be filled with in order to gain the requested size. (See the *Effect*.)
	`new_size`	The new size.
Postconditions:	`size() == new_size && capacity() >= new_size` If the new size is greater than the current size, copies of `item` will be inserted at the **back** of the of the `circular_buffer` in order to achieve the desired size. In the case the resulting size exceeds the current capacity the capacity will be set to `new_size`. If the current number of elements stored in the `circular_buffer` is greater than the desired new size then number of `[size() - new_size]` **last** elements will be removed. (The capacity will remain unchanged.)	
Throws:	An allocation error if memory is exhausted (`std::bad_alloc` if the standard allocator is used). Whatever `T::T(const T&)` throws or nothing if `T::T(T&&)` is noexcept.	

34.
```
void rset_capacity(capacity_type new_capacity);
```

Change the capacity of the `circular_buffer`.

Exception Safety. Strong.

Iterator Invalidation. Invalidates all iterators pointing to the `circular_buffer` (except iterators equal to `end()`) if the new capacity is different from the original.

Complexity. Linear (in `min[size(), new_capacity]`).

See Also:

`set_capacity(capacity_type)`, `rresize(size_type, const_reference)`

Parameters:	`new_capacity`	The new capacity.
Requires:	If `T` is a move only type, then compiler shall support `noexcept` modifiers and move constructor of `T` must be marked with it (must not throw exceptions).	
Postconditions:	`capacity() == new_capacity && size() <= new_capacity` If the current number of elements stored in the `circular_buffer` is greater than the desired new capacity then number of `[size() - new_capacity]` **first** elements will be removed and the new size will be equal to `new_capacity`.	
Throws:	An allocation error if memory is exhausted (`std::bad_alloc` if the standard allocator is used). Whatever `T::T(const T&)` throws or nothing if `T::T(T&&)` is noexcept.	

35.
```
void rresize(size_type new_size, param_value_type item = value_type());
```

Change the size of the `circular_buffer`.

Exception Safety. Basic.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Iterator Invalidation. Invalidates all iterators pointing to the `circular_buffer` (except iterators equal to `end()`) if the new size is greater than the current capacity. Invalidates iterators pointing to the removed elements if the new size is lower that the original size. Otherwise it does not invalidate any iterator.

Complexity. Linear (in the new size of the `circular_buffer`).

See Also:

`resize(size_type, const_reference)`, `rset_capacity(capacity_type)`

Parameters:	`item`	The element the `circular_buffer` will be filled with in order to gain the requested size. (See the *Effect*.)
	`new_size`	The new size.

Postconditions: `size() == new_size && capacity() >= new_size`
If the new size is greater than the current size, copies of `item` will be inserted at the **front** of the of the `circular_buffer` in order to achieve the desired size. In the case the resulting size exceeds the current capacity the capacity will be set to `new_size`.
If the current number of elements stored in the `circular_buffer` is greater than the desired new size then number of `[size() - new_size]` **first** elements will be removed. (The capacity will remain unchanged.)

Throws: An allocation error if memory is exhausted (`std::bad_alloc` if the standard allocator is used). Whatever `T::T(const T&)` throws or nothing if `T::T(T&&)` is noexcept.

36.
```
void assign(size_type n, param_value_type item);
```

Assign `n` items into the `circular_buffer`.

The content of the `circular_buffer` will be removed and replaced with `n` copies of the `item`.

Exception Safety. Basic.

Iterator Invalidation. Invalidates all iterators pointing to the `circular_buffer` (except iterators equal to `end()`).

Complexity. Linear (in the `n`).

See Also:

`operator=`, `assign(capacity_type, size_type, const_reference)`, `assign(InputIterator, InputIterator)`, `assign(capacity_type, InputIterator, InputIterator)`

Parameters:	`item`	The element the `circular_buffer` will be filled with.
	`n`	The number of elements the `circular_buffer` will be filled with.

Postconditions: `capacity() == n && size() == n && (*this)[0] == item && (*this)[1] == item && ... && (*this) [n - 1] == item`

Throws: An allocation error if memory is exhausted (`std::bad_alloc` if the standard allocator is used). Whatever `T::T(const T&)` throws.

37.
```
void assign(capacity_type buffer_capacity, size_type n, param_value_type item);
```

Assign `n` items into the `circular_buffer` specifying the capacity.

The capacity of the `circular_buffer` will be set to the specified value and the content of the `circular_buffer` will be removed and replaced with `n` copies of the `item`.

Exception Safety. Basic.

Iterator Invalidation. Invalidates all iterators pointing to the `circular_buffer` (except iterators equal to `end()`).

Complexity. Linear (in the `n`).

See Also:

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

`operator=`, `assign(size_type, const_reference)`, `assign(InputIterator, InputIterator)`, `assign(capacity_type, InputIterator, InputIterator)`

Parameters:	`buffer_capacity`	The new capacity.
	`item`	The element the `circular_buffer` will be filled with.
	`n`	The number of elements the `circular_buffer` will be filled with.
Requires:	`capacity >= n`	
Postconditions:	`capacity() == buffer_capacity && size() == n && (*this)[0] == item && (*this)[1] == item && ... && (*this) [n - 1] == item`	
Throws:	An allocation error if memory is exhausted (`std::bad_alloc` if the standard allocator is used). Whatever `T::T(const T&)` throws.	

38.
```
template<typename InputIterator>
  void assign(InputIterator first, InputIterator last);
```

Assign a copy of the range into the `circular_buffer`.

The content of the `circular_buffer` will be removed and replaced with copies of elements from the specified range.

Exception Safety. Basic.

Iterator Invalidation. Invalidates all iterators pointing to the `circular_buffer` (except iterators equal to `end()`).

Complexity. Linear (in the `std::distance(first, last)`).

See Also:

`operator=`, `assign(size_type, const_reference)`, `assign(capacity_type, size_type, const_reference)`, `assign(capacity_type, InputIterator, InputIterator)`

Parameters:	`first`	The beginning of the range to be copied.
	`last`	The end of the range to be copied.
Requires:	Valid range `[first, last)`. `first` and `last` have to meet the requirements of InputIterator.	
Postconditions:	`capacity() == std::distance(first, last) && size() == std::distance(first, last) && (*this)[0]== *first && (*this)[1] == *(first + 1) && ... && (*this)[std::distance(first, last) - 1] == *(last - 1)`	
Throws:	An allocation error if memory is exhausted (`std::bad_alloc` if the standard allocator is used). Whatever `T::T(const T&)` throws.	

39.
```
template<typename InputIterator>
  void assign(capacity_type buffer_capacity, InputIterator first,
              InputIterator last);
```

Assign a copy of the range into the `circular_buffer` specifying the capacity.

The capacity of the `circular_buffer` will be set to the specified value and the content of the `circular_buffer` will be removed and replaced with copies of elements from the specified range.

Exception Safety. Basic.

Iterator Invalidation. Invalidates all iterators pointing to the `circular_buffer` (except iterators equal to `end()`).

Complexity. Linear (in `std::distance(first, last)`; in `min[capacity, std::distance(first, last)]` if the `InputIterator` is a RandomAccessIterator).

See Also:

`operator=`, `assign(size_type, const_reference)`, `assign(capacity_type, size_type, const_reference)`, `assign(InputIterator, InputIterator)`

Parameters:	`buffer_capacity`	The new capacity.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

	`first` The beginning of the range to be copied. `last` The end of the range to be copied.
Requires:	Valid range `[first, last)`. `first` and `last` have to meet the requirements of InputIterator.
Postconditions:	`capacity() == buffer_capacity && size() <= std::distance(first, last) && (*this)[0]== *(last - buffer_capacity) && (*this)[1] == *(last - buffer_capacity + 1) && ... && (*this)[buffer_capacity - 1] == *(last - 1)` If the number of items to be copied from the range `[first, last)` is greater than the specified `buffer_capacity` then only elements from the range `[last - buffer_capacity, last)` will be copied.
Throws:	An allocation error if memory is exhausted (`std::bad_alloc` if the standard allocator is used). Whatever `T::T(const T&)` throws.

40.
```
void swap(circular_buffer< T, Alloc > & cb) noexcept;
```

Swap the contents of two `circular_buffer`s.

Exception Safety. No-throw.

Iterator Invalidation. Invalidates all iterators of both `circular_buffer`s. (On the other hand the iterators still point to the same elements but within another container. If you want to rely on this feature you have to turn the Debug Support off otherwise an assertion will report an error if such invalidated iterator is used.)

Complexity. Constant (in the size of the `circular_buffer`).

See Also:

`swap(circular_buffer<T, Alloc>&, circular_buffer<T, Alloc>&)`

Parameters:	`cb` The `circular_buffer` whose content will be swapped.
Postconditions:	`this` contains elements of `cb` and vice versa; the capacity of `this` equals to the capacity of `cb` and vice versa.
Throws:	Nothing.

41.
```
void push_back(param_value_type item);
```

Insert a new element at the end of the `circular_buffer`.

Exception Safety. Basic; no-throw if the operation in the *Throws* section does not throw anything.

Iterator Invalidation. Does not invalidate any iterators with the exception of iterators pointing to the overwritten element.

Complexity. Constant (in the size of the `circular_buffer`).

See Also:

`push_front(const_reference)`, `pop_back()`, `pop_front()`

Parameters:	`item` The element to be inserted.
Postconditions:	if `capacity() > 0` then `back() == item` If the `circular_buffer` is full, the first element will be removed. If the capacity is `0`, nothing will be inserted.
Throws:	Whatever `T::T(const T&)` throws. Whatever `T::operator = (const T&)` throws.

42.
```
void push_back(rvalue_type item);
```

Insert a new element at the end of the `circular_buffer` using rvalue references or rvalues references emulation.

Exception Safety. Basic; no-throw if the operation in the *Throws* section does not throw anything.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Iterator Invalidation. Does not invalidate any iterators with the exception of iterators pointing to the overwritten element.

Complexity. Constant (in the size of the `circular_buffer`).

See Also:

`push_front(const_reference)`, `pop_back()`, `pop_front()`

Parameters:	`item`	The element to be inserted.
Postconditions:	if `capacity() > 0` then `back() == item` If the `circular_buffer` is full, the first element will be removed. If the capacity is `0`, nothing will be inserted.	
Throws:	Whatever `T::T(T&&)` throws. Whatever `T::operator = (T&&)` throws.	

43.
```
void push_back();
```

Insert a new default-constructed element at the end of the `circular_buffer`.

Exception Safety. Basic; no-throw if the operation in the *Throws* section does not throw anything.

Iterator Invalidation. Does not invalidate any iterators with the exception of iterators pointing to the overwritten element.

Complexity. Constant (in the size of the `circular_buffer`).

See Also:

`push_front(const_reference)`, `pop_back()`, `pop_front()`

Postconditions:	if `capacity() > 0` then `back() == item` If the `circular_buffer` is full, the first element will be removed. If the capacity is `0`, nothing will be inserted.
Throws:	Whatever `T::T()` throws. Whatever `T::T(T&&)` throws. Whatever `T::operator = (T&&)` throws.

44.
```
void push_front(param_value_type item);
```

Insert a new element at the beginning of the `circular_buffer`.

Exception Safety. Basic; no-throw if the operation in the *Throws* section does not throw anything.

Iterator Invalidation. Does not invalidate any iterators with the exception of iterators pointing to the overwritten element.

Complexity. Constant (in the size of the `circular_buffer`).

See Also:

`push_back(const_reference)`, `pop_back()`, `pop_front()`

Parameters:	`item`	The element to be inserted.
Postconditions:	if `capacity() > 0` then `front() == item` If the `circular_buffer` is full, the last element will be removed. If the capacity is `0`, nothing will be inserted.	
Throws:	Whatever `T::T(const T&)` throws. Whatever `T::operator = (const T&)` throws.	

45.
```
void push_front(rvalue_type item);
```

Insert a new element at the beginning of the `circular_buffer` using rvalue references or rvalues references emulation.

Exception Safety. Basic; no-throw if the operation in the *Throws* section does not throw anything.

Iterator Invalidation. Does not invalidate any iterators with the exception of iterators pointing to the overwritten element.

Complexity. Constant (in the size of the `circular_buffer`).

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

See Also:

`push_back(const_reference)`, `pop_back()`, `pop_front()`

Parameters:	`item` The element to be inserted.
Postconditions:	if `capacity() > 0` then `front() == item` If the `circular_buffer` is full, the last element will be removed. If the capacity is `0`, nothing will be inserted.
Throws:	Whatever `T::T(T&&)` throws. Whatever `T::operator = (T&&)` throws.

46.
```
void push_front();
```

Insert a new default-constructed element at the beginning of the `circular_buffer`.

Exception Safety. Basic; no-throw if the operation in the *Throws* section does not throw anything.

Iterator Invalidation. Does not invalidate any iterators with the exception of iterators pointing to the overwritten element.

Complexity. Constant (in the size of the `circular_buffer`).

See Also:

`push_back(const_reference)`, `pop_back()`, `pop_front()`

Postconditions:	if `capacity() > 0` then `front() == item` If the `circular_buffer` is full, the last element will be removed. If the capacity is `0`, nothing will be inserted.
Throws:	Whatever `T::T()` throws. Whatever `T::T(T&&)` throws. Whatever `T::operator = (T&&)` throws.

47.
```
void pop_back();
```

Remove the last element from the `circular_buffer`.

Exception Safety. No-throw.

Iterator Invalidation. Invalidates only iterators pointing to the removed element.

Complexity. Constant (in the size of the `circular_buffer`).

See Also:

`pop_front()`, `push_back(const_reference)`, `push_front(const_reference)`

Requires:	`!empty()`
Postconditions:	The last element is removed from the `circular_buffer`.
Throws:	Nothing.

48.
```
void pop_front();
```

Remove the first element from the `circular_buffer`.

Exception Safety. No-throw.

Iterator Invalidation. Invalidates only iterators pointing to the removed element.

Complexity. Constant (in the size of the `circular_buffer`).

See Also:

`pop_back()`, `push_back(const_reference)`, `push_front(const_reference)`

Requires:	`!empty()`
Postconditions:	The first element is removed from the `circular_buffer`.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Throws:	Nothing.

49.

```
iterator insert(iterator pos, param_value_type item);
```

Insert an element at the specified position.

Exception Safety. Basic; no-throw if the operation in the *Throws* section does not throw anything.

Iterator Invalidation. Invalidates iterators pointing to the elements at the insertion point (including `pos`) and iterators behind the insertion point (towards the end; except iterators equal to `end()`). It also invalidates iterators pointing to the overwritten element.

Complexity. Linear (in `std::distance(pos, end())`).

See Also:

`insert(iterator, size_type, value_type)`, `insert(iterator, InputIterator, InputIterator)`, `rinsert(iterator, value_type)`, `rinsert(iterator, size_type, value_type)`, `rinsert(iterator, InputIterator, InputIterator)`

Parameters:	`item` The element to be inserted. `pos` An iterator specifying the position where the `item` will be inserted.
Requires:	`pos` is a valid iterator pointing to the `circular_buffer` or its end.
Postconditions:	The `item` will be inserted at the position `pos`. If the `circular_buffer` is full, the first element will be overwritten. If the `circular_buffer` is full and the `pos` points to `begin()`, then the `item` will not be inserted. If the capacity is `0`, nothing will be inserted.
Returns:	Iterator to the inserted element or `begin()` if the `item` is not inserted. (See the *Effect*.)
Throws:	Whatever `T::T(const T&)` throws. Whatever `T::operator = (const T&)` throws. Exceptions of move_if_noexcept(T&).

50.

```
iterator insert(iterator pos, rvalue_type item);
```

Insert an element at the specified position.

Exception Safety. Basic; no-throw if the operation in the *Throws* section does not throw anything.

Iterator Invalidation. Invalidates iterators pointing to the elements at the insertion point (including `pos`) and iterators behind the insertion point (towards the end; except iterators equal to `end()`). It also invalidates iterators pointing to the overwritten element.

Complexity. Linear (in `std::distance(pos, end())`).

See Also:

`insert(iterator, size_type, value_type)`, `insert(iterator, InputIterator, InputIterator)`, `rinsert(iterator, value_type)`, `rinsert(iterator, size_type, value_type)`, `rinsert(iterator, InputIterator, InputIterator)`

Parameters:	`item` The element to be inserted. `pos` An iterator specifying the position where the `item` will be inserted.
Requires:	`pos` is a valid iterator pointing to the `circular_buffer` or its end.
Postconditions:	The `item` will be inserted at the position `pos`. If the `circular_buffer` is full, the first element will be overwritten. If the `circular_buffer` is full and the `pos` points to `begin()`, then the `item` will not be inserted. If the capacity is `0`, nothing will be inserted.
Returns:	Iterator to the inserted element or `begin()` if the `item` is not inserted. (See the *Effect*.)
Throws:	Whatever `T::T(T&&)` throws. Whatever `T::operator = (T&&)` throws. Exceptions of move_if_noexcept(T&).

51.

```
iterator insert(iterator pos);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Insert a default-constructed element at the specified position.

Exception Safety. Basic; no-throw if the operation in the *Throws* section does not throw anything.

Iterator Invalidation. Invalidates iterators pointing to the elements at the insertion point (including `pos`) and iterators behind the insertion point (towards the end; except iterators equal to `end()`). It also invalidates iterators pointing to the overwritten element.

Complexity. Linear (in `std::distance(pos, end())`).

See Also:

`insert(iterator, size_type, value_type)`, `insert(iterator, InputIterator, InputIterator)`, `rinsert(iterator, value_type)`,`rinsert(iterator, size_type, value_type)`,`rinsert(iterator, InputIterator, InputIterator)`

Parameters:	`pos` An iterator specifying the position where the `item` will be inserted.
Requires:	`pos` is a valid iterator pointing to the `circular_buffer` or its end.
Postconditions:	The `item` will be inserted at the position `pos`. If the `circular_buffer` is full, the first element will be overwritten. If the `circular_buffer` is full and the `pos` points to `begin()`, then the `item` will not be inserted. If the capacity is `0`, nothing will be inserted.
Returns:	Iterator to the inserted element or `begin()` if the `item` is not inserted. (See the *Effect*.)
Throws:	Whatever `T::T()` throws. Whatever `T::T(T&&)` throws. Whatever `T::operator = (T&&)` throws. Exceptions of move_if_noexcept(T&).

52.

```
void insert(iterator pos, size_type n, param_value_type item);
```

Insert `n` copies of the `item` at the specified position.

Exception Safety. Basic; no-throw if the operations in the *Throws* section do not throw anything.

Iterator Invalidation. Invalidates iterators pointing to the elements at the insertion point (including `pos`) and iterators behind the insertion point (towards the end; except iterators equal to `end()`). It also invalidates iterators pointing to the overwritten elements.

Complexity. Linear (in `min[capacity(), std::distance(pos, end()) + n]`).

Example. Consider a `circular_buffer` with the capacity of 6 and the size of 4. Its internal buffer may look like the one below.
`|1|2|3|4| | |`
`p ___^`
After inserting 5 elements at the position `p`:
`insert(p, (size_t)5, 0);`
actually only 4 elements get inserted and elements `1` and `2` are overwritten. This is due to the fact the insert operation preserves the capacity. After insertion the internal buffer looks like this:
`|0|0|0|0|3|4|`
For comparison if the capacity would not be preserved the internal buffer would then result in `|1|2|0|0|0|0|0|3|4|`.

See Also:

`insert(iterator, value_type)`, `insert(iterator, InputIterator, InputIterator)`, `rinsert(iterator, value_type)`,`rinsert(iterator, size_type, value_type)`,`rinsert(iterator, InputIterator, InputIterator)`

Parameters:	`item` The element whose copies will be inserted. `n` The number of `item`s the to be inserted. `pos` An iterator specifying the position where the `item`s will be inserted.
Requires:	`pos` is a valid iterator pointing to the `circular_buffer` or its end.
Postconditions:	The number of `min[n, (pos - begin()) + reserve()]` elements will be inserted at the position `pos`. The number of `min[pos - begin(), max[0, n - reserve()]]` elements will be overwritten at the beginning of the `circular_buffer`.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

	(See *Example* for the explanation.)
Throws:	Whatever `T::T(const T&)` throws. Whatever `T::operator = (const T&)` throws. Exceptions of move_if_noexcept(T&).

53.
```
template<typename InputIterator>
  void insert(iterator pos, InputIterator first, InputIterator last);
```

Insert the range `[first, last)` at the specified position.

Exception Safety. Basic; no-throw if the operations in the *Throws* section do not throw anything.

Iterator Invalidation. Invalidates iterators pointing to the elements at the insertion point (including `pos`) and iterators behind the insertion point (towards the end; except iterators equal to `end()`). It also invalidates iterators pointing to the overwritten elements.

Complexity. Linear (in `[std::distance(pos, end()) + std::distance(first, last)]`; in `min[capacity(), std::distance(pos, end()) + std::distance(first, last)]` if the `InputIterator` is a RandomAccessIterator).

Example. Consider a `circular_buffer` with the capacity of 6 and the size of 4. Its internal buffer may look like the one below.

`|1|2|3|4| | |`

`p ___^`

After inserting a range of elements at the position `p`:

`int array[] = { 5, 6, 7, 8, 9 };`

`insert(p, array, array + 5);`

actually only elements `6`, `7`, `8` and `9` from the specified range get inserted and elements `1` and `2` are overwritten. This is due to the fact the insert operation preserves the capacity. After insertion the internal buffer looks like this:

`|6|7|8|9|3|4|`

For comparison if the capacity would not be preserved the internal buffer would then result in `|1|2|5|6|7|8|9|3|4|`.

See Also:

`insert(iterator, value_type)`, `insert(iterator, size_type, value_type)`, `rinsert(iterator, value_type)`, `rinsert(iterator, size_type, value_type)`, `rinsert(iterator, InputIterator, InputIterator)`

Parameters:	`first` The beginning of the range to be inserted. `last` The end of the range to be inserted. `pos` An iterator specifying the position where the range will be inserted.
Requires:	`pos` is a valid iterator pointing to the `circular_buffer` or its end. Valid range `[first, last)` where `first` and `last` meet the requirements of an InputIterator.
Postconditions:	Elements from the range `[first + max[0, distance(first, last) - (pos - begin()) - reserve()], last)` will be inserted at the position `pos`. The number of `min[pos - begin(), max[0, distance(first, last) - reserve()]]` elements will be overwritten at the beginning of the `circular_buffer`. (See *Example* for the explanation.)
Throws:	Whatever `T::T(const T&)` throws if the `InputIterator` is not a move iterator. Whatever `T::operator = (const T&)` throws if the `InputIterator` is not a move iterator. Whatever `T::T(T&&)` throws if the `InputIterator` is a move iterator. Whatever `T::operator = (T&&)` throws if the `InputIterator` is a move iterator.

54.
```
iterator rinsert(iterator pos, param_value_type item);
```

Insert an element before the specified position.

Exception Safety. Basic; no-throw if the operations in the *Throws* section do not throw anything.

Iterator Invalidation. Invalidates iterators pointing to the elements before the insertion point (towards the beginning and excluding `pos`). It also invalidates iterators pointing to the overwritten element.

Complexity. Linear (in `std::distance(begin(), pos)`).

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

See Also:

`rinsert(iterator, size_type, value_type)`, `rinsert(iterator, InputIterator, InputIterator)`, `insert(iterator, value_type)`, `insert(iterator, size_type, value_type)`, `insert(iterator, InputIterator, InputIterator)`

Parameters:	`item` The element to be inserted. `pos` An iterator specifying the position before which the `item` will be inserted.
Requires:	`pos` is a valid iterator pointing to the `circular_buffer` or its end.
Postconditions:	The `item` will be inserted before the position `pos`. If the `circular_buffer` is full, the last element will be overwritten. If the `circular_buffer` is full and the `pos` points to `end()`, then the `item` will not be inserted. If the capacity is `0`, nothing will be inserted.
Returns:	Iterator to the inserted element or `end()` if the `item` is not inserted. (See the *Effect*.)
Throws:	Whatever `T::T(const T&)` throws. Whatever `T::operator = (const T&)` throws. Exceptions of move_if_noexcept(T&).

55.
```
iterator rinsert(iterator pos, rvalue_type item);
```

Insert an element before the specified position.

Exception Safety. Basic; no-throw if the operations in the *Throws* section do not throw anything.

Iterator Invalidation. Invalidates iterators pointing to the elements before the insertion point (towards the beginning and excluding `pos`). It also invalidates iterators pointing to the overwritten element.

Complexity. Linear (in `std::distance(begin(), pos)`).

See Also:

`rinsert(iterator, size_type, value_type)`, `rinsert(iterator, InputIterator, InputIterator)`, `insert(iterator, value_type)`, `insert(iterator, size_type, value_type)`, `insert(iterator, InputIterator, InputIterator)`

Parameters:	`item` The element to be inserted. `pos` An iterator specifying the position before which the `item` will be inserted.
Requires:	`pos` is a valid iterator pointing to the `circular_buffer` or its end.
Postconditions:	The `item` will be inserted before the position `pos`. If the `circular_buffer` is full, the last element will be overwritten. If the `circular_buffer` is full and the `pos` points to `end()`, then the `item` will not be inserted. If the capacity is `0`, nothing will be inserted.
Returns:	Iterator to the inserted element or `end()` if the `item` is not inserted. (See the *Effect*.)
Throws:	Whatever `T::T(T&&)` throws. Whatever `T::operator = (T&&)` throws. Exceptions of move_if_noexcept(T&).

56.
```
iterator rinsert(iterator pos);
```

Insert an element before the specified position.

Exception Safety. Basic; no-throw if the operations in the *Throws* section do not throw anything.

Iterator Invalidation. Invalidates iterators pointing to the elements before the insertion point (towards the beginning and excluding `pos`). It also invalidates iterators pointing to the overwritten element.

Complexity. Linear (in `std::distance(begin(), pos)`).

See Also:

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

`rinsert(iterator, size_type, value_type)`, `rinsert(iterator, InputIterator, InputIterator)`, `insert(iterator, value_type)`, `insert(iterator, size_type, value_type)`, `insert(iterator, InputIterator, InputIterator)`

Parameters:	`pos` An iterator specifying the position before which the `item` will be inserted.
Requires:	`pos` is a valid iterator pointing to the `circular_buffer` or its end.
Postconditions:	The `item` will be inserted before the position `pos`. If the `circular_buffer` is full, the last element will be overwritten. If the `circular_buffer` is full and the `pos` points to `end()`, then the `item` will not be inserted. If the capacity is `0`, nothing will be inserted.
Returns:	Iterator to the inserted element or `end()` if the `item` is not inserted. (See the *Effect*.)
Throws:	Whatever `T::T()` throws. Whatever `T::T(T&&)` throws. Whatever `T::operator = (T&&)` throws. Exceptions of move_if_noexcept(T&).

57.
```
void rinsert(iterator pos, size_type n, param_value_type item);
```

Insert `n` copies of the `item` before the specified position.

Exception Safety. Basic; no-throw if the operations in the *Throws* section do not throw anything.

Iterator Invalidation. Invalidates iterators pointing to the elements before the insertion point (towards the beginning and excluding `pos`). It also invalidates iterators pointing to the overwritten elements.

Complexity. Linear (in `min[capacity(), std::distance(begin(), pos) + n]`).

Example. Consider a `circular_buffer` with the capacity of 6 and the size of 4. Its internal buffer may look like the one below.
`|1|2|3|4| | |`
`p ___^`
After inserting 5 elements before the position `p`:
`rinsert(p, (size_t)5, 0);`
actually only 4 elements get inserted and elements `3` and `4` are overwritten. This is due to the fact the rinsert operation preserves the capacity. After insertion the internal buffer looks like this:
`|1|2|0|0|0|0|`
For comparison if the capacity would not be preserved the internal buffer would then result in `|1|2|0|0|0|0|0|3|4|`.

See Also:

`rinsert(iterator, value_type)`, `rinsert(iterator, InputIterator, InputIterator)`, `insert(iterator, value_type)`, `insert(iterator, size_type, value_type)`, `insert(iterator, InputIterator, InputIterator)`

Parameters:	`item` The element whose copies will be inserted. `n` The number of `item`s the to be inserted. `pos` An iterator specifying the position where the `item`s will be inserted.
Requires:	`pos` is a valid iterator pointing to the `circular_buffer` or its end.
Postconditions:	The number of `min[n, (end() - pos) + reserve()]` elements will be inserted before the position `pos`. The number of `min[end() - pos, max[0, n - reserve()]]` elements will be overwritten at the end of the `circular_buffer`. (See *Example* for the explanation.)
Throws:	Whatever `T::T(const T&)` throws. Whatever `T::operator = (const T&)` throws. Exceptions of move_if_noexcept(T&).

58.
```
template<typename InputIterator>
  void rinsert(iterator pos, InputIterator first, InputIterator last);
```

Insert the range `[first, last)` before the specified position.

Exception Safety. Basic; no-throw if the operations in the *Throws* section do not throw anything.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Iterator Invalidation. Invalidates iterators pointing to the elements before the insertion point (towards the beginning and excluding `pos`). It also invalidates iterators pointing to the overwritten elements.

Complexity. Linear (in `[std::distance(begin(), pos) + std::distance(first, last)]`; in `min[capacity(), std::distance(begin(), pos) + std::distance(first, last)]` if the `InputIterator` is a RandomAccessIterator).

Example. Consider a `circular_buffer` with the capacity of 6 and the size of 4. Its internal buffer may look like the one below.
`|1|2|3|4| | |`
`p ___^`
After inserting a range of elements before the position `p`:
`int array[] = { 5, 6, 7, 8, 9 };`
`insert(p, array, array + 5);`
actually only elements `5`, `6`, `7` and `8` from the specified range get inserted and elements `3` and `4` are overwritten. This is due to the fact the rinsert operation preserves the capacity. After insertion the internal buffer looks like this:
`|1|2|5|6|7|8|`
For comparison if the capacity would not be preserved the internal buffer would then result in `|1|2|5|6|7|8|9|3|4|`.

See Also:

`rinsert(iterator, value_type)`, `rinsert(iterator, size_type, value_type)`, `insert(iterator, value_type)`, `insert(iterator, size_type, value_type)`, `insert(iterator, InputIterator, InputIterator)`

Parameters:	`first`	The beginning of the range to be inserted.
	`last`	The end of the range to be inserted.
	`pos`	An iterator specifying the position where the range will be inserted.

Requires:	`pos` is a valid iterator pointing to the `circular_buffer` or its end. Valid range `[first, last)` where `first` and `last` meet the requirements of an InputIterator.
Postconditions:	Elements from the range `[first, last - max[0, distance(first, last) - (end() - pos) - reserve()])` will be inserted before the position `pos`. The number of `min[end() - pos, max[0, distance(first, last) - reserve()]]` elements will be overwritten at the end of the `circular_buffer`. (See *Example* for the explanation.)
Throws:	Whatever `T::T(const T&)` throws if the `InputIterator` is not a move iterator. Whatever `T::operator = (const T&)` throws if the `InputIterator` is not a move iterator. Whatever `T::T(T&&)` throws if the `InputIterator` is a move iterator. Whatever `T::operator = (T&&)` throws if the `InputIterator` is a move iterator.

59.
```
iterator erase(iterator pos);
```

Remove an element at the specified position.

Exception Safety. Basic; no-throw if the operation in the *Throws* section does not throw anything.

Iterator Invalidation. Invalidates iterators pointing to the erased element and iterators pointing to the elements behind the erased element (towards the end; except iterators equal to `end()`).

Complexity. Linear (in `std::distance(pos, end())`).

See Also:

`erase(iterator, iterator)`, `rerase(iterator)`, `rerase(iterator, iterator)`, `erase_begin(size_type)`, `erase_end(size_type)`, `clear()`

Parameters:	`pos`	An iterator pointing at the element to be removed.

Requires:	`pos` is a valid iterator pointing to the `circular_buffer` (but not an `end()`).
Postconditions:	The element at the position `pos` is removed.
Returns:	Iterator to the first element remaining beyond the removed element or `end()` if no such element exists.
Throws:	<a href="circular_buffer/implementation.html#circular_buffer.implementation.exceptions_of_move_if_noexcept_t">Exceptions of move_if_noexcept(T&).

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

60.

```
iterator erase(iterator first, iterator last);
```

Erase the range `[first, last)`.

Exception Safety. Basic; no-throw if the operation in the *Throws* section does not throw anything.

Iterator Invalidation. Invalidates iterators pointing to the erased elements and iterators pointing to the elements behind the erased range (towards the end; except iterators equal to `end()`).

Complexity. Linear (in `std::distance(first, end())`).

See Also:

`erase(iterator)`, `rerase(iterator)`, `rerase(iterator, iterator)`, `erase_begin(size_type)`, `erase_end(size_type)`, `clear()`

Parameters:	`first`	The beginning of the range to be removed.
	`last`	The end of the range to be removed.
Requires:	Valid range `[first, last)`.	
Postconditions:	The elements from the range `[first, last)` are removed. (If `first == last` nothing is removed.)	
Returns:	Iterator to the first element remaining beyond the removed elements or `end()` if no such element exists.	
Throws:	<a href="circular_buffer/implementation.html#circular_buffer.implementation.exceptions_of_move_if_noexcept_t">Exceptions of move_if_noexcept(T&).	

61.

```
iterator rerase(iterator pos);
```

Remove an element at the specified position.

Exception Safety. Basic; no-throw if the operation in the *Throws* section does not throw anything.

Iterator Invalidation. Invalidates iterators pointing to the erased element and iterators pointing to the elements in front of the erased element (towards the beginning).

Complexity. Linear (in `std::distance(begin(), pos)`).

Note

This method is symetric to the `erase(iterator)` method and is more effective than `erase(iterator)` if the iterator `pos` is close to the beginning of the `circular_buffer`. (See the *Complexity*.)

See Also:

`erase(iterator)`, `erase(iterator, iterator)`, `rerase(iterator, iterator)`, `erase_begin(size_type)`, `erase_end(size_type)`, `clear()`

Parameters:	`pos`	An iterator pointing at the element to be removed.
Requires:	`pos` is a valid iterator pointing to the `circular_buffer` (but not an `end()`).	
Postconditions:	The element at the position `pos` is removed.	
Returns:	Iterator to the first element remaining in front of the removed element or `begin()` if no such element exists.	
Throws:	<a href="circular_buffer/implementation.html#circular_buffer.implementation.exceptions_of_move_if_noexcept_t">Exceptions of move_if_noexcept(T&).	

62.

```
iterator rerase(iterator first, iterator last);
```

Erase the range `[first, last)`.

Exception Safety. Basic; no-throw if the operation in the *Throws* section does not throw anything.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Iterator Invalidation. Invalidates iterators pointing to the erased elements and iterators pointing to the elements in front of the erased range (towards the beginning).

Complexity. Linear (in `std::distance(begin(), last)`).

> ## Note
>
> This method is symetric to the `erase(iterator, iterator)` method and is more effective than `erase(iterator, iterator)` if `std::distance(begin(), first)` is lower that `std::distance(last, end())`.

See Also:

`erase(iterator)`, `erase(iterator, iterator)`, `rerase(iterator)`, `erase_begin(size_type)`, `erase_end(size_type)`, `clear()`

Parameters:	`first` The beginning of the range to be removed. `last` The end of the range to be removed.
Requires:	Valid range `[first, last)`.
Postconditions:	The elements from the range `[first, last)` are removed. (If `first == last` nothing is removed.)
Returns:	Iterator to the first element remaining in front of the removed elements or `begin()` if no such element exists.
Throws:	<a href="circular_buffer/implementation.html#circular_buffer.implementation.exceptions_of_move_if_noexcept_t">Exceptions of move_if_noexcept(T&).

63.

```
void erase_begin(size_type n);
```

Remove first `n` elements (with constant complexity for scalar types).

Exception Safety. Basic; no-throw if the operation in the *Throws* section does not throw anything. (I.e. no throw in case of scalars.)

Iterator Invalidation. Invalidates iterators pointing to the first `n` erased elements.

Complexity. Constant (in `n`) for scalar types; linear for other types.

> ## Note
>
> This method has been specially designed for types which do not require an explicit destructruction (e.g. integer, float or a pointer). For these scalar types a call to a destructor is not required which makes it possible to implement the "erase from beginning" operation with a constant complexity. For non-sacalar types the complexity is linear (hence the explicit destruction is needed) and the implementation is actually equivalent to `rerase(begin(), begin() + n)`.

See Also:

`erase(iterator)`, `erase(iterator, iterator)`, `rerase(iterator)`, `rerase(iterator, iterator)`, `erase_end(size_type)`, `clear()`

Parameters:	`n` The number of elements to be removed.
Requires:	`n <= size()`
Postconditions:	The `n` elements at the beginning of the `circular_buffer` will be removed.
Throws:	<a href="circular_buffer/implementation.html#circular_buffer.implementation.exceptions_of_move_if_noexcept_t">Exceptions of move_if_noexcept(T&).

64.

```
void erase_end(size_type n);
```

Remove last `n` elements (with constant complexity for scalar types).

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Exception Safety. Basic; no-throw if the operation in the *Throws* section does not throw anything. (I.e. no throw in case of scalars.)

Iterator Invalidation. Invalidates iterators pointing to the last `n` erased elements.

Complexity. Constant (in `n`) for scalar types; linear for other types.

> ### Note
>
> This method has been specially designed for types which do not require an explicit destructruction (e.g. integer, float or a pointer). For these scalar types a call to a destructor is not required which makes it possible to implement the "erase from end" operation with a constant complexity. For non-sacalar types the complexity is linear (hence the explicit destruction is needed) and the implementation is actually equivalent to `erase(end() - n, end())`.

See Also:

`erase(iterator)`, `erase(iterator, iterator)`, `rerase(iterator)`, `rerase(iterator, iterator)`, `erase_begin(size_type)`, `clear()`

Parameters:	`n` The number of elements to be removed.
Requires:	`n <= size()`
Postconditions:	The `n` elements at the end of the `circular_buffer` will be removed.
Throws:	<a href="circular_buffer/implementation.html#circular_buffer.implementation.exceptions_of_move_if_noexcept_t">Exceptions of move_if_noexcept(T&).

65.
```
void clear() noexcept;
```

Remove all stored elements from the `circular_buffer`.

Exception Safety. No-throw.

Iterator Invalidation. Invalidates all iterators pointing to the `circular_buffer` (except iterators equal to `end()`).

Complexity. Constant (in the size of the `circular_buffer`) for scalar types; linear for other types.

See Also:

`~circular_buffer()`, `erase(iterator)`, `erase(iterator, iterator)`, `rerase(iterator)`, `rerase(iterator, iterator)`, `erase_begin(size_type)`, `erase_end(size_type)`

Postconditions:	`size() == 0`
Throws:	Nothing.

`circular_buffer` private member functions

1.
```
template<typename ValT> void push_back_impl(ValT item);
```

2.
```
template<typename ValT> void push_front_impl(ValT item);
```

3.
```
template<typename ValT> iterator insert_impl(iterator pos, ValT item);
```

4.
```
template<typename ValT> iterator rinsert_impl(iterator pos, ValT item);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

5.

```
void check_position(size_type index) const;
```

Check if the `index` is valid.

6.

```
template<typename Pointer> void increment(Pointer & p) const;
```

Increment the pointer.

7.

```
template<typename Pointer> void decrement(Pointer & p) const;
```

Decrement the pointer.

8.

```
template<typename Pointer> Pointer add(Pointer p, difference_type n) const;
```

Add `n` to the pointer.

9.

```
template<typename Pointer> Pointer sub(Pointer p, difference_type n) const;
```

Subtract `n` from the pointer.

10.

```
pointer map_pointer(pointer p) const;
```

Map the null pointer to virtual end of circular buffer.

11.

```
pointer allocate(size_type n);
```

Allocate memory.

12.

```
void deallocate(pointer p, size_type n);
```

Deallocate memory.

13.

```
bool is_uninitialized(const_pointer p) const noexcept;
```

Does the pointer point to the uninitialized memory?

14.

```
void replace(pointer pos, param_value_type item);
```

Replace an element.

15.

```
void replace(pointer pos, rvalue_type item);
```

Replace an element.

16.

```
void construct_or_replace(bool construct, pointer pos, param_value_type item);
```

Construct or replace an element.

`construct` has to be set to `true` if and only if `pos` points to an uninitialized memory.

17.

```
void construct_or_replace(bool construct, pointer pos, rvalue_type item);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Construct or replace an element.

`construct` has to be set to `true` if and only if `pos` points to an uninitialized memory.

18.
```
void destroy_item(pointer p);
```

Destroy an item.

19.
```
void destroy_if_constructed(pointer pos);
```

Destroy an item only if it has been constructed.

20.
```
void destroy_content();
```

Destroy the whole content of the circular buffer.

21.
```
void destroy_content(const true_type &);
```

Specialized destroy_content method.

22.
```
void destroy_content(const false_type &);
```

Specialized destroy_content method.

23.
```
void destroy() noexcept;
```

Destroy content and free allocated memory.

24.
```
void initialize_buffer(capacity_type buffer_capacity);
```

Initialize the internal buffer.

25.
```
void initialize_buffer(capacity_type buffer_capacity, param_value_type item);
```

Initialize the internal buffer.

26.
```
template<typename IntegralType> 
  void initialize(IntegralType n, IntegralType item, const true_type &);
```

Specialized initialize method.

27.
```
template<typename Iterator> 
  void initialize(Iterator first, Iterator last, const false_type &);
```

Specialized initialize method.

28.
```
template<typename InputIterator> 
  void initialize(InputIterator first, InputIterator last, 
                  const std::input_iterator_tag &);
```

Specialized initialize method.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

29.
```
template<typename ForwardIterator> 
  void initialize(ForwardIterator first, ForwardIterator last, 
                  const std::forward_iterator_tag &);
```

Specialized initialize method.

30.
```
template<typename IntegralType> 
  void initialize(capacity_type buffer_capacity, IntegralType n, 
                  IntegralType item, const true_type &);
```

Specialized initialize method.

31.
```
template<typename Iterator> 
  void initialize(capacity_type buffer_capacity, Iterator first, 
                  Iterator last, const false_type &);
```

Specialized initialize method.

32.
```
template<typename InputIterator> 
  void initialize(capacity_type buffer_capacity, InputIterator first, 
                  InputIterator last, const std::input_iterator_tag &);
```

Specialized initialize method.

33.
```
template<typename ForwardIterator> 
  void initialize(capacity_type buffer_capacity, ForwardIterator first, 
                  ForwardIterator last, const std::forward_iterator_tag &);
```

Specialized initialize method.

34.
```
template<typename ForwardIterator> 
  void initialize(capacity_type buffer_capacity, ForwardIterator first, 
                  ForwardIterator last, size_type distance);
```

Initialize the circular buffer.

35.
```
void reset(pointer buff, pointer last, capacity_type new_capacity);
```

Reset the circular buffer.

36.
```
void swap_allocator(circular_buffer< T, Alloc > &, const true_type &);
```

Specialized method for swapping the allocator.

37.
```
void swap_allocator(circular_buffer< T, Alloc > & cb, const false_type &);
```

Specialized method for swapping the allocator.

38.
```
template<typename IntegralType> 
  void assign(IntegralType n, IntegralType item, const true_type &);
```

Specialized assign method.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

39.

```
template<typename Iterator>
  void assign(Iterator first, Iterator last, const false_type &);
```

Specialized assign method.

40.

```
template<typename InputIterator>
  void assign(InputIterator first, InputIterator last,
              const std::input_iterator_tag &);
```

Specialized assign method.

41.

```
template<typename ForwardIterator>
  void assign(ForwardIterator first, ForwardIterator last,
              const std::forward_iterator_tag &);
```

Specialized assign method.

42.

```
template<typename IntegralType>
  void assign(capacity_type new_capacity, IntegralType n, IntegralType item,
              const true_type &);
```

Specialized assign method.

43.

```
template<typename Iterator>
  void assign(capacity_type new_capacity, Iterator first, Iterator last,
              const false_type &);
```

Specialized assign method.

44.

```
template<typename InputIterator>
  void assign(capacity_type new_capacity, InputIterator first,
              InputIterator last, const std::input_iterator_tag &);
```

Specialized assign method.

45.

```
template<typename ForwardIterator>
  void assign(capacity_type new_capacity, ForwardIterator first,
              ForwardIterator last, const std::forward_iterator_tag &);
```

Specialized assign method.

46.

```
template<typename Functor>
  void assign_n(capacity_type new_capacity, size_type n, const Functor & fnc);
```

Helper assign method.

47.

```
template<typename ValT> iterator insert_item(const iterator & pos, ValT item);
```

Helper insert method.

48.

```
template<typename IntegralType>
  void insert(const iterator & pos, IntegralType n, IntegralType item,
              const true_type &);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Specialized insert method.

49.
```
template<typename Iterator>
  void insert(const iterator & pos, Iterator first, Iterator last,
              const false_type &);
```

Specialized insert method.

50.
```
template<typename InputIterator>
  void insert(iterator pos, InputIterator first, InputIterator last,
              const std::input_iterator_tag &);
```

Specialized insert method.

51.
```
template<typename ForwardIterator>
  void insert(const iterator & pos, ForwardIterator first,
              ForwardIterator last, const std::forward_iterator_tag &);
```

Specialized insert method.

52.
```
template<typename Wrapper>
  void insert_n(const iterator & pos, size_type n, const Wrapper & wrapper);
```

Helper insert method.

53.
```
template<typename IntegralType>
  void rinsert(const iterator & pos, IntegralType n, IntegralType item,
               const true_type &);
```

Specialized rinsert method.

54.
```
template<typename Iterator>
  void rinsert(const iterator & pos, Iterator first, Iterator last,
               const false_type &);
```

Specialized rinsert method.

55.
```
template<typename InputIterator>
  void rinsert(iterator pos, InputIterator first, InputIterator last,
               const std::input_iterator_tag &);
```

Specialized insert method.

56.
```
template<typename ForwardIterator>
  void rinsert(const iterator & pos, ForwardIterator first,
               ForwardIterator last, const std::forward_iterator_tag &);
```

Specialized rinsert method.

57.
```
template<typename Wrapper>
  void rinsert_n(const iterator & pos, size_type n, const Wrapper & wrapper);
```

Helper rinsert method.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

58.
```
void erase_begin(size_type n, const true_type &);
```

Specialized erase_begin method.

59.
```
void erase_begin(size_type n, const false_type &);
```

Specialized erase_begin method.

60.
```
void erase_end(size_type n, const true_type &);
```

Specialized erase_end method.

61.
```
void erase_end(size_type n, const false_type &);
```

Specialized erase_end method.

Function template operator==

boost::operator== — Compare two `circular_buffer`s element-by-element to determine if they are equal.

Synopsis

```
// In header: <boost/circular_buffer/base.hpp>


template<typename T, typename Alloc>
  bool operator==(const circular_buffer< T, Alloc > & lhs,
                  const circular_buffer< T, Alloc > & rhs);
```

Description

Complexity. Linear (in the size of the `circular_buffer`s).

Iterator Invalidation. Does not invalidate any iterators.

Parameters:	`lhs`	The `circular_buffer` to compare.
	`rhs`	The `circular_buffer` to compare.
Returns:	`lhs.size() == rhs.size() && std::equal(lhs.begin(), lhs.end(), rhs.begin())`	
Throws:	Nothing.	

Function template operator<

boost::operator< — Compare two `circular_buffer`s element-by-element to determine if the left one is lesser than the right one.

Synopsis

```
// In header: <boost/circular_buffer/base.hpp>


template<typename T, typename Alloc>
  bool operator<(const circular_buffer< T, Alloc > & lhs,
                 const circular_buffer< T, Alloc > & rhs);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Description

Complexity. Linear (in the size of the `circular_buffer`s).

Iterator Invalidation. Does not invalidate any iterators.

Parameters:	`lhs`	The `circular_buffer` to compare.
	`rhs`	The `circular_buffer` to compare.
Returns:	`std::lexicographical_compare(lhs.begin(), lhs.end(), rhs.begin(), rhs.end())`	
Throws:	Nothing.	

Function template operator!=

boost::operator!= — Compare two `circular_buffer`s element-by-element to determine if they are non-equal.

Synopsis

```
// In header: <boost/circular_buffer/base.hpp>


template<typename T, typename Alloc>
  bool operator!=(const circular_buffer< T, Alloc > & lhs,
                  const circular_buffer< T, Alloc > & rhs);
```

Description

Complexity. Linear (in the size of the `circular_buffer`s).

Iterator Invalidation. Does not invalidate any iterators.

See Also:

`operator==(const circular_buffer<T,Alloc>&, const circular_buffer<T,Alloc>&)`

Parameters:	`lhs`	The `circular_buffer` to compare.
	`rhs`	The `circular_buffer` to compare.
Returns:	`!(lhs == rhs)`	
Throws:	Nothing.	

Function template operator>

boost::operator> — Compare two `circular_buffer`s element-by-element to determine if the left one is greater than the right one.

Synopsis

```
// In header: <boost/circular_buffer/base.hpp>


template<typename T, typename Alloc>
  bool operator>(const circular_buffer< T, Alloc > & lhs,
                 const circular_buffer< T, Alloc > & rhs);
```

Description

Complexity. Linear (in the size of the `circular_buffer`s).

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Iterator Invalidation. Does not invalidate any iterators.

See Also:

`operator<(const circular_buffer<T,Alloc>&, const circular_buffer<T,Alloc>&)`

Parameters:	`lhs`	The `circular_buffer` to compare.
	`rhs`	The `circular_buffer` to compare.
Returns:	`rhs < lhs`	
Throws:	Nothing.	

Function template operator<=

boost::operator<= — Compare two `circular_buffer`s element-by-element to determine if the left one is lesser or equal to the right one.

Synopsis

```
// In header: <boost/circular_buffer/base.hpp>


template<typename T, typename Alloc> 
  bool operator<=(const circular_buffer< T, Alloc > & lhs, 
                  const circular_buffer< T, Alloc > & rhs);
```

Description

Complexity. Linear (in the size of the `circular_buffer`s).

Iterator Invalidation. Does not invalidate any iterators.

See Also:

`operator<(const circular_buffer<T,Alloc>&, const circular_buffer<T,Alloc>&)`

Parameters:	`lhs`	The `circular_buffer` to compare.
	`rhs`	The `circular_buffer` to compare.
Returns:	`!(rhs < lhs)`	
Throws:	Nothing.	

Function template operator>=

boost::operator>= — Compare two `circular_buffer`s element-by-element to determine if the left one is greater or equal to the right one.

Synopsis

```
// In header: <boost/circular_buffer/base.hpp>


template<typename T, typename Alloc> 
  bool operator>=(const circular_buffer< T, Alloc > & lhs, 
                  const circular_buffer< T, Alloc > & rhs);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Description

Complexity. Linear (in the size of the `circular_buffer`s).

Iterator Invalidation. Does not invalidate any iterators.

See Also:

`operator<(const circular_buffer<T,Alloc>&, const circular_buffer<T,Alloc>&)`

Parameters:	`lhs`	The `circular_buffer` to compare.
	`rhs`	The `circular_buffer` to compare.
Returns:	`!(lhs < rhs)`	
Throws:	Nothing.	

Function template swap

boost::swap — Swap the contents of two `circular_buffer`s.

Synopsis

```
// In header: <boost/circular_buffer/base.hpp>


template<typename T, typename Alloc>
  void swap(circular_buffer< T, Alloc > & lhs,
            circular_buffer< T, Alloc > & rhs);
```

Description

Complexity. Constant (in the size of the `circular_buffer`s).

Iterator Invalidation. Invalidates all iterators of both `circular_buffer`s. (On the other hand the iterators still point to the same elements but within another container. If you want to rely on this feature you have to turn the Debug Support off otherwise an assertion will report an error if such invalidated iterator is used.)

See Also:

`swap(circular_buffer<T, Alloc>&)`

Parameters:	`lhs`	The `circular_buffer` whose content will be swapped with `rhs`.
	`rhs`	The `circular_buffer` whose content will be swapped with `lhs`.
Postconditions:	`lhs` contains elements of `rhs` and vice versa.	
Throws:	Nothing.	

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Header <boost/circular_buffer/debug.hpp>

Header <boost/circular_buffer/details.hpp>

Header <boost/circular_buffer/space_optimized.hpp>

```
namespace boost {
  template<typename T, typename Alloc> class circular_buffer_space_optimized;

  // Test two space optimized circular buffers for equality.
  template<typename T, typename Alloc>
    bool operator==(const circular_buffer_space_optimized< T, Alloc > & lhs,
                    const circular_buffer_space_optimized< T, Alloc > & rhs);

  // Lexicographical comparison.
  template<typename T, typename Alloc>
    bool operator<(const circular_buffer_space_optimized< T, Alloc > & lhs,
                   const circular_buffer_space_optimized< T, Alloc > & rhs);

  // Test two space optimized circular buffers for non-equality.
  template<typename T, typename Alloc>
    bool operator!=(const circular_buffer_space_optimized< T, Alloc > & lhs,
                    const circular_buffer_space_optimized< T, Alloc > & rhs);

  // Lexicographical comparison.
  template<typename T, typename Alloc>
    bool operator>(const circular_buffer_space_optimized< T, Alloc > & lhs,
                   const circular_buffer_space_optimized< T, Alloc > & rhs);

  // Lexicographical comparison.
  template<typename T, typename Alloc>
    bool operator<=(const circular_buffer_space_optimized< T, Alloc > & lhs,
                    const circular_buffer_space_optimized< T, Alloc > & rhs);

  // Lexicographical comparison.
  template<typename T, typename Alloc>
    bool operator>=(const circular_buffer_space_optimized< T, Alloc > & lhs,
                    const circular_buffer_space_optimized< T, Alloc > & rhs);

  // Swap the contents of two space optimized circular buffers.
  template<typename T, typename Alloc>
    void swap(circular_buffer_space_optimized< T, Alloc > & lhs,
              circular_buffer_space_optimized< T, Alloc > & rhs);
}
```

Class template circular_buffer_space_optimized

boost::circular_buffer_space_optimized — Space optimized circular buffer container adaptor. `T` must be a copyable class or must have an noexcept move constructor and move assignment operator.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/circular_buffer/space_optimized.hpp>

template<typename T, typename Alloc>
class circular_buffer_space_optimized :
  private boost::circular_buffer< T, Alloc >
{
public:
  // types
  typedef circular_buffer< T, Alloc >::value_type             value_type;
  typedef circular_buffer< T, Alloc >::pointer                pointer;
  typedef circular_buffer< T, Alloc >::const_pointer          const_pointer;
  typedef circular_buffer< T, Alloc >::reference              reference;
  typedef circular_buffer< T, Alloc >::const_reference        const_reference;
  typedef circular_buffer< T, Alloc >::size_type              size_type;
  typedef circular_buffer< T, Alloc >::difference_type        difference_type;
  typedef circular_buffer< T, Alloc >::allocator_type         allocator_type;
  typedef circular_buffer< T, Alloc >::const_iterator         const_iterator;
  typedef circular_buffer< T, Alloc >::iterator               iterator;
  typedef circular_buffer< T, Alloc >::const_reverse_iterator const_reverse_iterator;
  typedef circular_buffer< T, Alloc >::reverse_iterator       reverse_iterator;
  typedef circular_buffer< T, Alloc >::array_range            array_range;
  typedef circular_buffer< T, Alloc >::const_array_range      const_array_range;
  typedef circular_buffer< T, Alloc >::param_value_type       param_value_type;
  typedef circular_buffer< T, Alloc >::rvalue_type            rvalue_type;
  typedef cb_details::capacity_control< size_type >           capacity_type;

  // construct/copy/destruct
  explicit circular_buffer_space_optimized(const allocator_type & = allocator_type()) noexcept;
  explicit circular_buffer_space_optimized(capacity_type,
                                           const allocator_type & = allocator_type());
  circular_buffer_space_optimized(capacity_type, param_value_type,
                                  const allocator_type & = allocator_type());
  circular_buffer_space_optimized(capacity_type, size_type, param_value_type,
                                  const allocator_type & = allocator_type());
  circular_buffer_space_optimized(const circular_buffer_space_optimized< T, Alloc > &);
  circular_buffer_space_optimized(circular_buffer_space_optimized< T, Alloc > &&) noexcept;
  template<typename InputIterator>
    circular_buffer_space_optimized(InputIterator, InputIterator,
                                    const allocator_type & = allocator_type());
  template<typename InputIterator>
    circular_buffer_space_optimized(capacity_type, InputIterator,
                                    InputIterator,
                                    const allocator_type & = allocator_type());
  circular_buffer_space_optimized< T, Alloc > &
  operator=(const circular_buffer_space_optimized< T, Alloc > &);
  circular_buffer_space_optimized< T, Alloc > &
  operator=(circular_buffer_space_optimized< T, Alloc > &&) noexcept;

  // public member functions
  bool full() const noexcept;
  size_type reserve() const noexcept;
  const capacity_type & capacity() const noexcept;
  void set_capacity(const capacity_type &);
  void resize(size_type, param_value_type = value_type());
  void rset_capacity(const capacity_type &);
  void rresize(size_type, param_value_type = value_type());
  void assign(size_type, param_value_type);
  void assign(capacity_type, size_type, param_value_type);
  template<typename InputIterator> void assign(InputIterator, InputIterator);
  template<typename InputIterator>
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
  void assign(capacity_type, InputIterator, InputIterator);
void swap(circular_buffer_space_optimized< T, Alloc > &) noexcept;
void push_back(param_value_type);
void push_back(rvalue_type);
void push_back();
void push_front(param_value_type);
void push_front(rvalue_type);
void push_front();
void pop_back();
void pop_front();
iterator insert(iterator, param_value_type);
iterator insert(iterator, rvalue_type);
iterator insert(iterator);
void insert(iterator, size_type, param_value_type);
template<typename InputIterator>
  void insert(iterator, InputIterator, InputIterator);
iterator rinsert(iterator, param_value_type);
iterator rinsert(iterator, rvalue_type);
iterator rinsert(iterator);
void rinsert(iterator, size_type, param_value_type);
template<typename InputIterator>
  void rinsert(iterator, InputIterator, InputIterator);
iterator erase(iterator);
iterator erase(iterator, iterator);
iterator rerase(iterator);
iterator rerase(iterator, iterator);
void clear();

// private member functions
void adjust_min_capacity();
size_type ensure_reserve(size_type, size_type) const;
void check_low_capacity(size_type = 1);
void check_high_capacity();
void reduce_capacity(const true_type &);
void reduce_capacity(const false_type &);
template<typename IntegralType>
  void insert(const iterator &, IntegralType, IntegralType,
              const true_type &);
template<typename Iterator>
  void insert(const iterator &, Iterator, Iterator, const false_type &);
template<typename IntegralType>
  void rinsert(const iterator &, IntegralType, IntegralType,
               const true_type &);
template<typename Iterator>
  void rinsert(const iterator &, Iterator, Iterator, const false_type &);

// private static functions
static size_type init_capacity(const capacity_type &, size_type);
template<typename IntegralType>
  static size_type
  init_capacity(const capacity_type &, IntegralType, IntegralType,
                const true_type &);
template<typename Iterator>
  static size_type
  init_capacity(const capacity_type &, Iterator, Iterator,
                const false_type &);
template<typename InputIterator>
  static size_type
  init_capacity(const capacity_type &, InputIterator, InputIterator,
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
                    const std::input_iterator_tag &);
  template<typename ForwardIterator> 
    static size_type 
    init_capacity(const capacity_type &, ForwardIterator, ForwardIterator, 
                  const std::forward_iterator_tag &);
};
```

Description

`circular_buffer_space_optimized` public types

1. typedef cb_details::capacity_control< size_type > capacity_type;

   Capacity controller of the space optimized circular buffer.

   See Also:

   capacity_control in details.hpp.

   ```
   class capacity_control
   {
   size_type m_capacity; // Available capacity.
   size_type m_min_capacity; // Minimum capacity.
   public:
   capacity_control(size_type capacity, size_type min_capacity = 0)
   : m_capacity(capacity), m_min_capacity(min_capacity)
   {};
   size_type capacity() const { return m_capacity; }
   size_type min_capacity() const { return m_min_capacity; }
   operator size_type() const { return m_capacity; }
   };
   ```

   Always `capacity >= min_capacity`.

   The `capacity()` represents the capacity of the `circular_buffer_space_optimized` and the `min_capacity()` determines the minimal allocated size of its internal buffer.

   The converting constructor of the `capacity_control` allows implicit conversion from `size_type`-like types which ensures compatibility of creating an instance of the `circular_buffer_space_optimized` with other STL containers.

   On the other hand the operator `size_type()` provides implicit conversion to the `size_type` which allows to treat the capacity of the `circular_buffer_space_optimized` the same way as in the `circular_buffer`.

`circular_buffer_space_optimized` public construct/copy/destruct

1.
   ```
   explicit circular_buffer_space_optimized(const allocator_type & alloc = allocator_type()) no↵
   except;
   ```

   Create an empty space optimized circular buffer with zero capacity.

   Complexity. Constant.

   > **Warning**
   >
   > Since Boost version 1.36 the behaviour of this constructor has changed. Now it creates a space optimized circular buffer with zero capacity.

Parameters:	`alloc`	The allocator.
Postconditions:	`capacity().capacity() == 0 && capacity().min_capacity() == 0 && size() == 0`	

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Throws: Nothing.

2.

```
explicit circular_buffer_space_optimized(capacity_type capacity_ctrl,
                                         const allocator_type & alloc = allocator_type());
```

Create an empty space optimized circular buffer with the specified capacity.

Complexity. Constant.

Parameters:	`alloc`	The allocator.
	`capacity_ctrl`	The capacity controller representing the maximum number of elements which can be stored in the `circular_buffer_space_optimized` and the minimal allocated size of the internal buffer.
Postconditions:	`capacity() == capacity_ctrl && size() == 0` The amount of allocated memory in the internal buffer is `capacity_ctrl.min_capacity()`.	
Throws:	An allocation error if memory is exhausted (`std::bad_alloc` if the standard allocator is used).	

3.

```
circular_buffer_space_optimized(capacity_type capacity_ctrl,
                                param_value_type item,
                                const allocator_type & alloc = allocator_type());
```

Create a full space optimized circular buffer with the specified capacity filled with `capacity_ctrl.capacity()` copies of `item`.

Complexity. Linear (in the `capacity_ctrl.capacity()`).

Parameters:	`alloc`	The allocator.
	`capacity_ctrl`	The capacity controller representing the maximum number of elements which can be stored in the `circular_buffer_space_optimized` and the minimal allocated size of the internal buffer.
	`item`	The element the created `circular_buffer_space_optimized` will be filled with.
Postconditions:	`capacity() == capacity_ctrl && full() && (*this)[0] == item && (*this)[1] == item && ... && (*this) [capacity_ctrl.capacity() - 1] == item` The amount of allocated memory in the internal buffer is `capacity_ctrl.capacity()`.	
Throws:	An allocation error if memory is exhausted (`std::bad_alloc` if the standard allocator is used). `T::T(const T&)` throws.	

4.

```
circular_buffer_space_optimized(capacity_type capacity_ctrl, size_type n,
                                param_value_type item,
                                const allocator_type & alloc = allocator_type());
```

Create a space optimized circular buffer with the specified capacity filled with `n` copies of `item`.

Complexity. Linear (in the `n`).

Parameters:	`alloc`	The allocator.
	`capacity_ctrl`	The capacity controller representing the maximum number of elements which can be stored in the `circular_buffer_space_optimized` and the minimal allocated size of the internal buffer.
	`item`	The element the created `circular_buffer_space_optimized` will be filled with.
	`n`	The number of elements the created `circular_buffer_space_optimized` will be filled with.
Requires:	`capacity_ctrl.capacity() >= n`	
Postconditions:	`capacity() == capacity_ctrl && size() == n && (*this)[0] == item && (*this)[1] == item && ... && (*this)[n - 1] == item` The amount of allocated memory in the internal buffer is `max[n, capacity_ctrl.min_capacity()]`.	
Throws:	An allocation error if memory is exhausted (`std::bad_alloc` if the standard allocator is used). Whatever `T::T(const T&)` throws.	

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

5.
```
circular_buffer_space_optimized(const circular_buffer_space_optimized< T, Alloc > & cb);
```

The copy constructor.

Creates a copy of the specified `circular_buffer_space_optimized`.

Complexity. Linear (in the size of `cb`).

Parameters:	`cb`	The `circular_buffer_space_optimized` to be copied.
Postconditions:	`*this == cb` The amount of allocated memory in the internal buffer is `cb.size()`.	
Throws:	An allocation error if memory is exhausted (`std::bad_alloc` if the standard allocator is used). Whatever `T::T(const T&)` throws.	

6.
```
circular_buffer_space_optimized(circular_buffer_space_optimized< T, Alloc > && cb) noexcept;
```

The move constructor.

Move constructs a `circular_buffer_space_optimized` from `cb`, leaving `cb` empty.

Constant.

Parameters:	`cb`	`circular_buffer` to 'steal' value from.
Requires:	C++ compiler with rvalue references support.	
Postconditions:	`cb.empty()`	
Throws:	Nothing.	

7.
```
template<typename InputIterator>
  circular_buffer_space_optimized(InputIterator first, InputIterator last,
                                  const allocator_type & alloc = allocator_type());
```

Create a full space optimized circular buffer filled with a copy of the range.

Complexity. Linear (in the `std::distance(first, last)`).

Parameters:	`alloc`	The allocator.
	`first`	The beginning of the range to be copied.
	`last`	The end of the range to be copied.
Requires:	Valid range `[first, last)`. `first` and `last` have to meet the requirements of InputIterator.	
Postconditions:	`capacity().capacity() == std::distance(first, last) && capacity().min_capacity() == 0 && full() && (*this)[0]== *first && (*this)[1] == *(first + 1) && ... && (*this)[std::distance(first, last) - 1] == *(last - 1)` The amount of allocated memory in the internal buffer is `std::distance(first, last)`.	
Throws:	An allocation error if memory is exhausted (`std::bad_alloc` if the standard allocator is used). Whatever `T::T(const T&)` throws or nothing if `T::T(T&&)` is noexcept and `InputIterator` is a move iterator.	

8.
```
template<typename InputIterator>
  circular_buffer_space_optimized(capacity_type capacity_ctrl,
                                  InputIterator first, InputIterator last,
                                  const allocator_type & alloc = allocator_type());
```

Create a space optimized circular buffer with the specified capacity (and the minimal guaranteed amount of allocated memory) filled with a copy of the range.

Complexity. Linear (in `std::distance(first, last)`; in `min[capacity_ctrl.capacity(), std::distance(first, last)]` if the `InputIterator` is a RandomAccessIterator).

Parameters:	`alloc`	The allocator.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

	`capacity_ctrl`	The capacity controller representing the maximum number of elements which can be stored in the `circular_buffer_space_optimized` and the minimal allocated size of the internal buffer.
	`first`	The beginning of the range to be copied.
	`last`	The end of the range to be copied.

Requires: Valid range `[first, last)`.
`first` and `last` have to meet the requirements of InputIterator.

Postconditions: `capacity() == capacity_ctrl && size() <= std::distance(first, last) && (*this)[0]== (last - capacity_ctrl.capacity()) && (*this)[1] == *(last - capacity_ctrl.capacity() + 1) && ... && (*this)[capacity_ctrl.capacity() - 1] == *(last - 1)`
If the number of items to be copied from the range `[first, last)` is greater than the specified `capacity_ctrl.capacity()` then only elements from the range `[last - capacity_ctrl.capacity(), last)` will be copied.
The amount of allocated memory in the internal buffer is `max[capacity_ctrl.min_capacity(), min[capacity_ctrl.capacity(), std::distance(first, last)]]`.

Throws: An allocation error if memory is exhausted (`std::bad_alloc` if the standard allocator is used). Whatever `T::T(const T&)` throws.

9.
```
circular_buffer_space_optimized< T, Alloc > &
operator=(const circular_buffer_space_optimized< T, Alloc > & cb);
```

The assign operator.

Makes this `circular_buffer_space_optimized` to become a copy of the specified `circular_buffer_space_optimized`.

Exception Safety. Strong.

Iterator Invalidation. Invalidates all iterators pointing to this `circular_buffer_space_optimized` (except iterators equal to `end()`).

Complexity. Linear (in the size of `cb`).

See Also:

`assign(size_type, const_reference)`, `assign(capacity_type, size_type, const_reference)`, `assign(InputIterator, InputIterator)`, `assign(capacity_type, InputIterator, InputIterator)`

Parameters: `cb` The `circular_buffer_space_optimized` to be copied.

Postconditions: `*this == cb`
The amount of allocated memory in the internal buffer is `cb.size()`.

Throws: An allocation error if memory is exhausted (`std::bad_alloc` if the standard allocator is used). `T::T(const T&)` throws.

10.
```
circular_buffer_space_optimized< T, Alloc > &
operator=(circular_buffer_space_optimized< T, Alloc > && cb) noexcept;
```

Move assigns content of `cb` to `*this`, leaving `cb` empty.

Complexity. Constant.

Parameters: `cb` `circular_buffer` to 'steal' value from.
Requires: C++ compiler with rvalue references support.
Postconditions: `cb.empty()`
Throws: Nothing.

`circular_buffer_space_optimized` public member functions

1.
```
bool full() const noexcept;
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Is the `circular_buffer_space_optimized` full?

Exception Safety. No-throw.

Iterator Invalidation. Does not invalidate any iterators.

Complexity. Constant (in the size of the `circular_buffer_space_optimized`).

See Also:

`empty()`
Returns: `true` if the number of elements stored in the `circular_buffer_space_optimized` equals the capacity of the `circular_buffer_space_optimized`; `false` otherwise.
Throws: Nothing.

2.
```
size_type reserve() const noexcept;
```

Get the maximum number of elements which can be inserted into the `circular_buffer_space_optimized` without overwriting any of already stored elements.

Exception Safety. No-throw.

Iterator Invalidation. Does not invalidate any iterators.

Complexity. Constant (in the size of the `circular_buffer_space_optimized`).

See Also:

`capacity()`, `size()`, `max_size()`
Returns: `capacity().capacity() - size()`
Throws: Nothing.

3.
```
const capacity_type & capacity() const noexcept;
```

Get the capacity of the `circular_buffer_space_optimized`.

Exception Safety. No-throw.

Iterator Invalidation. Does not invalidate any iterators.

Complexity. Constant (in the size of the `circular_buffer_space_optimized`).

See Also:

`reserve()`, `size()`, `max_size()`, `set_capacity(const capacity_type&)`
Returns: The capacity controller representing the maximum number of elements which can be stored in the `circular_buffer_space_optimized` and the minimal allocated size of the internal buffer.
Throws: Nothing.

4.
```
void set_capacity(const capacity_type & capacity_ctrl);
```

Change the capacity (and the minimal guaranteed amount of allocated memory) of the `circular_buffer_space_optimized`.

Exception Safety. Strong.

Iterator Invalidation. Invalidates all iterators pointing to the `circular_buffer_space_optimized` (except iterators equal to `end()`).

Complexity. Linear (in `min[size(), capacity_ctrl.capacity()]`).

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Note

To explicitly clear the extra allocated memory use the **shrink-to-fit** technique:

```
boost::circular_buffer_space_optimized<int> cb(1000);
...
boost::circular_buffer_space_optimized<int>(cb).swap(cb);
```

For more information about the shrink-to-fit technique in STL see http://www.gotw.ca/gotw/054.htm.

See Also:

`rset_capacity(const capacity_type&)`, `resize(size_type, const_reference)`

Parameters:	`capacity_ctrl` The new capacity controller.
Postconditions:	`capacity() == capacity_ctrl && size() <= capacity_ctrl.capacity()` If the current number of elements stored in the `circular_buffer_space_optimized` is greater than the desired new capacity then number of `[size() - capacity_ctrl.capacity()]` **last** elements will be removed and the new size will be equal to `capacity_ctrl.capacity()`. If the current number of elements stored in the `circular_buffer_space_optimized` is lower than the new capacity then the amount of allocated memory in the internal buffer may be accommodated as necessary but it will never drop below `capacity_ctrl.min_capacity()`.
Throws:	An allocation error if memory is exhausted, (`std::bad_alloc` if the standard allocator is used). Whatever `T::T(const T&)` throws or nothing if `T::T(T&&)` is noexcept.

5.

```
void resize(size_type new_size, param_value_type item = value_type());
```

Change the size of the `circular_buffer_space_optimized`.

Exception Safety. Basic.

Iterator Invalidation. Invalidates all iterators pointing to the `circular_buffer_space_optimized` (except iterators equal to `end()`).

Complexity. Linear (in the new size of the `circular_buffer_space_optimized`).

See Also:

`rresize(size_type, const_reference)`, `set_capacity(const capacity_type&)`

Parameters:	`item` The element the `circular_buffer_space_optimized` will be filled with in order to gain the requested size. (See the *Effect*.) `new_size` The new size.
Postconditions:	`size() == new_size && capacity().capacity() >= new_size` If the new size is greater than the current size, copies of `item` will be inserted at the **back** of the of the `circular_buffer_space_optimized` in order to achieve the desired size. In the case the resulting size exceeds the current capacity the capacity will be set to `new_size`. If the current number of elements stored in the `circular_buffer_space_optimized` is greater than the desired new size then number of `[size() - new_size]` **last** elements will be removed. (The capacity will remain unchanged.) The amount of allocated memory in the internal buffer may be accommodated as necessary.
Throws:	An allocation error if memory is exhausted (`std::bad_alloc` if the standard allocator is used). Whatever `T::T(const T&)` throws.

6.

```
void rset_capacity(const capacity_type & capacity_ctrl);
```

Change the capacity (and the minimal guaranteed amount of allocated memory) of the `circular_buffer_space_optimized`.

Exception Safety. Strong.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Iterator Invalidation. Invalidates all iterators pointing to the `circular_buffer_space_optimized` (except iterators equal to `end()`).

Complexity. Linear (in `min[size(), capacity_ctrl.capacity()]`).

See Also:

`set_capacity(const capacity_type&)`, `rresize(size_type, const_reference)`

Parameters:	`capacity_ctrl`	The new capacity controller.
Postconditions:	`capacity() == capacity_ctrl && size() <= capacity_ctrl` If the current number of elements stored in the `circular_buffer_space_optimized` is greater than the desired new capacity then number of `[size() - capacity_ctrl.capacity()]` **first** elements will be removed and the new size will be equal to `capacity_ctrl.capacity()`. If the current number of elements stored in the `circular_buffer_space_optimized` is lower than the new capacity then the amount of allocated memory in the internal buffer may be accommodated as necessary but it will never drop below `capacity_ctrl.min_capacity()`.	
Throws:	An allocation error if memory is exhausted (`std::bad_alloc` if the standard allocator is used). Whatever `T::T(const T&)` throws or nothing if `T::T(T&&)` is noexcept.	

7.

```
void rresize(size_type new_size, param_value_type item = value_type());
```

Change the size of the `circular_buffer_space_optimized`.

Exception Safety. Basic.

Iterator Invalidation. Invalidates all iterators pointing to the `circular_buffer_space_optimized` (except iterators equal to `end()`).

Complexity. Linear (in the new size of the `circular_buffer_space_optimized`).

See Also:

`resize(size_type, const_reference)`, `rset_capacity(const capacity_type&)`

Parameters:	`item`	The element the `circular_buffer_space_optimized` will be filled with in order to gain the requested size. (See the *Effect*.)
	`new_size`	The new size.
Postconditions:	`size() == new_size && capacity().capacity() >= new_size` If the new size is greater than the current size, copies of `item` will be inserted at the **front** of the of the `circular_buffer_space_optimized` in order to achieve the desired size. In the case the resulting size exceeds the current capacity the capacity will be set to `new_size`. If the current number of elements stored in the `circular_buffer_space_optimized` is greater than the desired new size then number of `[size() - new_size]` **first** elements will be removed. (The capacity will remain unchanged.) The amount of allocated memory in the internal buffer may be accommodated as necessary.	
Throws:	An allocation error if memory is exhausted (`std::bad_alloc` if the standard allocator is used). Whatever `T::T(const T&)` throws.	

8.

```
void assign(size_type n, param_value_type item);
```

Assign `n` items into the space optimized circular buffer.

The content of the `circular_buffer_space_optimized` will be removed and replaced with `n` copies of the `item`.

Exception Safety. Basic.

Iterator Invalidation. Invalidates all iterators pointing to the `circular_buffer_space_optimized` (except iterators equal to `end()`).

Complexity. Linear (in the `n`).

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

See Also:

`operator=`, `assign(capacity_type, size_type, const_reference)`, `assign(InputIterator, InputIterator)`, `assign(capacity_type, InputIterator, InputIterator)`

Parameters:	`item`	The element the `circular_buffer_space_optimized` will be filled with.
	`n`	The number of elements the `circular_buffer_space_optimized` will be filled with.
Postconditions:	`capacity().capacity() == n && capacity().min_capacity() == 0 && size() == n && (*this)[0] == item && (*this)[1] == item && ... && (*this) [n - 1] == item` The amount of allocated memory in the internal buffer is `n`.	
Throws:	An allocation error if memory is exhausted (`std::bad_alloc` if the standard allocator is used). Whatever `T::T(const T&)` throws.	

9.
```
void assign(capacity_type capacity_ctrl, size_type n, param_value_type item);
```

Assign `n` items into the space optimized circular buffer specifying the capacity.

The capacity of the `circular_buffer_space_optimized` will be set to the specified value and the content of the `circular_buffer_space_optimized` will be removed and replaced with `n` copies of the `item`.

Exception Safety. Basic.

Iterator Invalidation. Invalidates all iterators pointing to the `circular_buffer_space_optimized` (except iterators equal to `end()`).

Complexity. Linear (in the `n`).

See Also:

`operator=`, `assign(size_type, const_reference)`, `assign(InputIterator, InputIterator)`, `assign(capacity_type, InputIterator, InputIterator)`

Parameters:	`capacity_ctrl`	The new capacity controller.
	`item`	The element the `circular_buffer_space_optimized` will be filled with.
	`n`	The number of elements the `circular_buffer_space_optimized` will be filled with.
Requires:	`capacity_ctrl.capacity() >= n`	
Postconditions:	`capacity() == capacity_ctrl && size() == n && (*this)[0] == item && (*this)[1] == item && ... && (*this) [n - 1] == item` The amount of allocated memory will be `max[n, capacity_ctrl.min_capacity()]`.	
Throws:	An allocation error if memory is exhausted (`std::bad_alloc` if the standard allocator is used). Whatever `T::T(const T&)` throws.	

10.
```
template<typename InputIterator>
  void assign(InputIterator first, InputIterator last);
```

Assign a copy of the range into the space optimized circular buffer.

The content of the `circular_buffer_space_optimized` will be removed and replaced with copies of elements from the specified range.

Exception Safety. Basic.

Iterator Invalidation. Invalidates all iterators pointing to the `circular_buffer_space_optimized` (except iterators equal to `end()`).

Complexity. Linear (in the `std::distance(first, last)`).

See Also:

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

`operator=`, `assign(size_type, const_reference)`, `assign(capacity_type, size_type, const_reference)`, `assign(capacity_type, InputIterator, InputIterator)`

Parameters:	`first` The beginning of the range to be copied. `last` The end of the range to be copied.
Requires:	Valid range `[first, last)`. `first` and `last` have to meet the requirements of InputIterator.
Postconditions:	`capacity().capacity() == std::distance(first, last) && capacity().min_capacity() == 0 && size() == std::distance(first, last) && (*this)[0]== *first && (*this)[1] == *(first + 1) && ... && (*this)[std::distance(first, last) - 1] == *(last - 1)` The amount of allocated memory in the internal buffer is `std::distance(first, last)`.
Throws:	An allocation error if memory is exhausted (`std::bad_alloc` if the standard allocator is used). Whatever `T::T(const T&)` throws or nothing if `T::T(T&&)` is noexcept and `InputIterator` is a move iterator.

11.

```
template<typename InputIterator>
  void assign(capacity_type capacity_ctrl, InputIterator first,
              InputIterator last);
```

Assign a copy of the range into the space optimized circular buffer specifying the capacity.

The capacity of the `circular_buffer_space_optimized` will be set to the specified value and the content of the `circular_buffer_space_optimized` will be removed and replaced with copies of elements from the specified range.

Exception Safety. Basic.

Iterator Invalidation. Invalidates all iterators pointing to the `circular_buffer_space_optimized` (except iterators equal to `end()`).

Complexity. Linear (in `std::distance(first, last)`; in `min[capacity_ctrl.capacity(), std::distance(first, last)]` if the `InputIterator` is a RandomAccessIterator).

See Also:

`operator=`, `assign(size_type, const_reference)`, `assign(capacity_type, size_type, const_reference)`, `assign(InputIterator, InputIterator)`

Parameters:	`capacity_ctrl` The new capacity controller. `first` The beginning of the range to be copied. `last` The end of the range to be copied.
Requires:	Valid range `[first, last)`. `first` and `last` have to meet the requirements of InputIterator.
Postconditions:	`capacity() == capacity_ctrl && size() <= std::distance(first, last) && (*this)[0]== *(last - capacity) && (*this)[1] == *(last - capacity + 1) && ... && (*this)[capacity - 1] == *(last - 1)` If the number of items to be copied from the range `[first, last)` is greater than the specified `capacity` then only elements from the range `[last - capacity, last)` will be copied. The amount of allocated memory in the internal buffer is `max[std::distance(first, last), capacity_ctrl.min_capacity()]`.
Throws:	An allocation error if memory is exhausted (`std::bad_alloc` if the standard allocator is used). Whatever `T::T(const T&)` throws or nothing if `T::T(T&&)` is noexcept and `InputIterator` is a move iterator.

12.

```
void swap(circular_buffer_space_optimized< T, Alloc > & cb) noexcept;
```

Swap the contents of two space-optimized circular-buffers.

Exception Safety. No-throw.

Iterator Invalidation. Invalidates all iterators of both `circular_buffer_space_optimized` containers. (On the other hand the iterators still point to the same elements but within another container. If you want to rely on this feature you have to turn the

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

__debug_support off by defining macro BOOST_CB_DISABLE_DEBUG, otherwise an assertion will report an error if such invalidated iterator is used.)

Complexity. Constant (in the size of the `circular_buffer_space_optimized`).

See Also:

`swap(circular_buffer<T, Alloc>&, circular_buffer<T, Alloc>&)`, `swap(circular_buffer_space_optimized<T, Alloc>&, circular_buffer_space_optimized<T, Alloc>&)`

Parameters:	`cb` The `circular_buffer_space_optimized` whose content will be swapped.
Postconditions:	`this` contains elements of `cb` and vice versa; the capacity and the amount of allocated memory in the internal buffer of `this` equal to the capacity and the amount of allocated memory of `cb` and vice versa.
Throws:	Nothing.

13.
```
void push_back(param_value_type item);
```

Insert a new element at the end of the space optimized circular buffer.

Exception Safety. Basic.

Iterator Invalidation. Invalidates all iterators pointing to the `circular_buffer_space_optimized` (except iterators equal to `end()`).

Complexity. Linear (in the size of the `circular_buffer_space_optimized`).

See Also:

`push_front(const_reference)`, `pop_back()`, `pop_front()`

Parameters:	`item` The element to be inserted.
Postconditions:	if `capacity().capacity() > 0` then `back() == item` If the `circular_buffer_space_optimized` is full, the first element will be removed. If the capacity is `0`, nothing will be inserted. The amount of allocated memory in the internal buffer may be predictively increased.
Throws:	An allocation error if memory is exhausted (`std::bad_alloc` if the standard allocator is used). Whatever `T::T(const T&)` throws.

14.
```
void push_back(rvalue_type item);
```

Insert a new element at the end of the space optimized circular buffer.

Exception Safety. Basic.

Iterator Invalidation. Invalidates all iterators pointing to the `circular_buffer_space_optimized` (except iterators equal to `end()`).

Complexity. Linear (in the size of the `circular_buffer_space_optimized`).

See Also:

`push_front(const_reference)`, `pop_back()`, `pop_front()`

Parameters:	`item` The element to be inserted.
Postconditions:	if `capacity().capacity() > 0` then `back() == item` If the `circular_buffer_space_optimized` is full, the first element will be removed. If the capacity is `0`, nothing will be inserted. The amount of allocated memory in the internal buffer may be predictively increased.
Throws:	An allocation error if memory is exhausted (`std::bad_alloc` if the standard allocator is used).

15.
```
void push_back();
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Insert a new element at the end of the space optimized circular buffer.

Exception Safety. Basic.

Iterator Invalidation. Invalidates all iterators pointing to the `circular_buffer_space_optimized` (except iterators equal to `end()`).

Complexity. Linear (in the size of the `circular_buffer_space_optimized`).

See Also:

`push_front(const_reference)`, `pop_back()`, `pop_front()`

Postconditions:	if `capacity().capacity() > 0` then `back() == item` If the `circular_buffer_space_optimized` is full, the first element will be removed. If the capacity is `0`, nothing will be inserted. The amount of allocated memory in the internal buffer may be predictively increased.
Throws:	An allocation error if memory is exhausted (`std::bad_alloc` if the standard allocator is used). Whatever `T::T()` throws. Whatever `T::T(const T&)` throws or nothing if `T::T(T&&)` is noexcept.

16.
```
void push_front(param_value_type item);
```

Insert a new element at the beginning of the space optimized circular buffer.

Exception Safety. Basic.

Iterator Invalidation. Invalidates all iterators pointing to the `circular_buffer_space_optimized` (except iterators equal to `end()`).

Complexity. Linear (in the size of the `circular_buffer_space_optimized`).

See Also:

`push_back(const_reference)`, `pop_back()`, `pop_front()`

Parameters:	`item` The element to be inserted.
Postconditions:	if `capacity().capacity() > 0` then `front() == item` If the `circular_buffer_space_optimized` is full, the last element will be removed. If the capacity is `0`, nothing will be inserted. The amount of allocated memory in the internal buffer may be predictively increased.
Throws:	An allocation error if memory is exhausted (`std::bad_alloc` if the standard allocator is used). Whatever `T::T(const T&)` throws.

17.
```
void push_front(rvalue_type item);
```

Insert a new element at the beginning of the space optimized circular buffer.

Exception Safety. Basic.

Iterator Invalidation. Invalidates all iterators pointing to the `circular_buffer_space_optimized` (except iterators equal to `end()`).

Complexity. Linear (in the size of the `circular_buffer_space_optimized`).

See Also:

`push_back(const_reference)`, `pop_back()`, `pop_front()`

Parameters:	`item` The element to be inserted.
Postconditions:	if `capacity().capacity() > 0` then `front() == item` If the `circular_buffer_space_optimized` is full, the last element will be removed. If the capacity is `0`, nothing will be inserted.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

The amount of allocated memory in the internal buffer may be predictively increased.

Throws: An allocation error if memory is exhausted (`std::bad_alloc` if the standard allocator is used). Whatever `T::T(const T&)` throws or nothing if `T::T(T&&)` is noexcept.

18.

```
void push_front();
```

Insert a new element at the beginning of the space optimized circular buffer.

Exception Safety. Basic.

Iterator Invalidation. Invalidates all iterators pointing to the `circular_buffer_space_optimized` (except iterators equal to `end()`).

Complexity. Linear (in the size of the `circular_buffer_space_optimized`).

See Also:

`push_back(const_reference)`, `pop_back()`, `pop_front()`

Postconditions: if `capacity().capacity() > 0` then `front() == item`
If the `circular_buffer_space_optimized` is full, the last element will be removed. If the capacity is `0`, nothing will be inserted.
The amount of allocated memory in the internal buffer may be predictively increased.

Throws: An allocation error if memory is exhausted (`std::bad_alloc` if the standard allocator is used). Whatever `T::T()` throws. Whatever `T::T(const T&)` throws or nothing if `T::T(T&&)` is noexcept.

19.

```
void pop_back();
```

Remove the last element from the space optimized circular buffer.

Exception Safety. Basic.

Iterator Invalidation. Invalidates all iterators pointing to the `circular_buffer_space_optimized` (except iterators equal to `end()`).

Complexity. Linear (in the size of the `circular_buffer_space_optimized`).

See Also:

`pop_front()`, `push_back(const_reference)`, `push_front(const_reference)`

Requires: `!empty()`

Postconditions: The last element is removed from the `circular_buffer_space_optimized`.
The amount of allocated memory in the internal buffer may be predictively decreased.

Throws: An allocation error if memory is exhausted (`std::bad_alloc` if the standard allocator is used).

20.

```
void pop_front();
```

Remove the first element from the space optimized circular buffer.

Exception Safety. Basic.

Iterator Invalidation. Invalidates all iterators pointing to the `circular_buffer_space_optimized` (except iterators equal to `end()`).

Complexity. Linear (in the size of the `circular_buffer_space_optimized`).

See Also:

`pop_back()`, `push_back(const_reference)`, `push_front(const_reference)`

Requires: `!empty()`

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Postconditions:	The first element is removed from the `circular_buffer_space_optimized`. The amount of allocated memory in the internal buffer may be predictively decreased.
Throws:	An allocation error if memory is exhausted (`std::bad_alloc` if the standard allocator is used).

21.
```
iterator insert(iterator pos, param_value_type item);
```

Insert an element at the specified position.

Exception Safety. Basic.

Iterator Invalidation. Invalidates all iterators pointing to the `circular_buffer_space_optimized` (except iterators equal to `end()`).

Complexity. Linear (in the size of the `circular_buffer_space_optimized`).

See Also:

`insert(iterator, size_type, value_type)`, `insert(iterator, InputIterator, InputIterator)`, `rinsert(iterator, value_type)`,`rinsert(iterator, size_type, value_type)`,`rinsert(iterator, InputIterator, InputIterator)`

Parameters:	`item`	The element to be inserted.
	`pos`	An iterator specifying the position where the `item` will be inserted.
Requires:	`pos` is a valid iterator pointing to the `circular_buffer_space_optimized` or its end.	
Postconditions:	The `item` will be inserted at the position `pos`. If the `circular_buffer_space_optimized` is full, the first element will be overwritten. If the `circular_buffer_space_optimized` is full and the `pos` points to `begin()`, then the `item` will not be inserted. If the capacity is `0`, nothing will be inserted. The amount of allocated memory in the internal buffer may be predictively increased.	
Returns:	Iterator to the inserted element or `begin()` if the `item` is not inserted. (See the *Effect*.)	
Throws:	An allocation error if memory is exhausted (`std::bad_alloc` if the standard allocator is used). Whatever `T::T(const T&)` throws. Whatever `T::operator = (const T&)` throws.	

22.
```
iterator insert(iterator pos, rvalue_type item);
```

Insert an element at the specified position.

Exception Safety. Basic.

Iterator Invalidation. Invalidates all iterators pointing to the `circular_buffer_space_optimized` (except iterators equal to `end()`).

Complexity. Linear (in the size of the `circular_buffer_space_optimized`).

See Also:

`insert(iterator, size_type, value_type)`, `insert(iterator, InputIterator, InputIterator)`, `rinsert(iterator, value_type)`,`rinsert(iterator, size_type, value_type)`,`rinsert(iterator, InputIterator, InputIterator)`

Parameters:	`item`	The element to be inserted.
	`pos`	An iterator specifying the position where the `item` will be inserted.
Requires:	`pos` is a valid iterator pointing to the `circular_buffer_space_optimized` or its end.	
Postconditions:	The `item` will be inserted at the position `pos`. If the `circular_buffer_space_optimized` is full, the first element will be overwritten. If the `circular_buffer_space_optimized` is full and the `pos` points to `begin()`, then the `item` will not be inserted. If the capacity is `0`, nothing will be inserted. The amount of allocated memory in the internal buffer may be predictively increased.	
Returns:	Iterator to the inserted element or `begin()` if the `item` is not inserted. (See the *Effect*.)	

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Throws:	An allocation error if memory is exhausted (`std::bad_alloc` if the standard allocator is used). Whatever `T::T(const T&)` throws or nothing if `T::T(T&&)` is noexcept.

23.
```
iterator insert(iterator pos);
```

Insert an element at the specified position.

Exception Safety. Basic.

Iterator Invalidation. Invalidates all iterators pointing to the `circular_buffer_space_optimized` (except iterators equal to `end()`).

Complexity. Linear (in the size of the `circular_buffer_space_optimized`).

See Also:

`insert(iterator, size_type, value_type)`, `insert(iterator, InputIterator, InputIterator)`, `rinsert(iterator, value_type)`,`rinsert(iterator, size_type, value_type)`,`rinsert(iterator, InputIterator, InputIterator)`

Parameters:	`pos` An iterator specifying the position where the `item` will be inserted.
Requires:	`pos` is a valid iterator pointing to the `circular_buffer_space_optimized` or its end.
Postconditions:	The `item` will be inserted at the position `pos`. If the `circular_buffer_space_optimized` is full, the first element will be overwritten. If the `circular_buffer_space_optimized` is full and the `pos` points to `begin()`, then the `item` will not be inserted. If the capacity is `0`, nothing will be inserted. The amount of allocated memory in the internal buffer may be predictively increased.
Returns:	Iterator to the inserted element or `begin()` if the `item` is not inserted. (See the *Effect*.)
Throws:	An allocation error if memory is exhausted (`std::bad_alloc` if the standard allocator is used). Whatever `T::T()` throws. Whatever `T::T(const T&)` throws or nothing if `T::T(T&&)` is noexcept.

24.
```
void insert(iterator pos, size_type n, param_value_type item);
```

Insert `n` copies of the `item` at the specified position.

Exception Safety. Basic.

Iterator Invalidation. Invalidates all iterators pointing to the `circular_buffer_space_optimized` (except iterators equal to `end()`).

Complexity. Linear (in `min[capacity().capacity(), size() + n]`).

Example. Consider a `circular_buffer_space_optimized` with the capacity of 6 and the size of 4. Its internal buffer may look like the one below.
`|1|2|3|4| | |`
`p ___^`
After inserting 5 elements at the position `p`:
`insert(p, (size_t)5, 0);`
actually only 4 elements get inserted and elements `1` and `2` are overwritten. This is due to the fact the insert operation preserves the capacity. After insertion the internal buffer looks like this:
`|0|0|0|0|3|4|`
For comparison if the capacity would not be preserved the internal buffer would then result in `|1|2|0|0|0|0|0|3|4|`.

See Also:

`insert(iterator, value_type)`, `insert(iterator, InputIterator, InputIterator)`, `rinsert(iterator, value_type)`,`rinsert(iterator, size_type, value_type)`,`rinsert(iterator, InputIterator, InputIterator)`

Parameters:	`item` The element whose copies will be inserted.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

	`n` The number of `items` the to be inserted. `pos` An iterator specifying the position where the `items` will be inserted.
Requires:	`pos` is a valid iterator pointing to the `circular_buffer_space_optimized` or its end.
Postconditions:	The number of `min[n, (pos - begin()) + reserve()]` elements will be inserted at the position `pos`. The number of `min[pos - begin(), max[0, n - reserve()]]` elements will be overwritten at the beginning of the `circular_buffer_space_optimized`. (See *Example* for the explanation.) The amount of allocated memory in the internal buffer may be predictively increased.
Throws:	An allocation error if memory is exhausted (`std::bad_alloc` if the standard allocator is used). Whatever `T::T(const T&)` throws. Whatever `T::operator = (const T&)` throws.

25.
```
template<typename InputIterator>
  void insert(iterator pos, InputIterator first, InputIterator last);
```

Insert the range `[first, last)` at the specified position.

Exception Safety. Basic.

Iterator Invalidation. Invalidates all iterators pointing to the `circular_buffer_space_optimized` (except iterators equal to `end()`).

Complexity. Linear (in `[size() + std::distance(first, last)]`; in `min[capacity().capacity(), size() + std::distance(first, last)]` if the `InputIterator` is a RandomAccessIterator).

Example. Consider a `circular_buffer_space_optimized` with the capacity of 6 and the size of 4. Its internal buffer may look like the one below.
`|1|2|3|4| | |`
`p ___^`
After inserting a range of elements at the position `p`:
`int array[] = { 5, 6, 7, 8, 9 };`
`insert(p, array, array + 5);`
actually only elements `6`, `7`, `8` and `9` from the specified range get inserted and elements `1` and `2` are overwritten. This is due to the fact the insert operation preserves the capacity. After insertion the internal buffer looks like this:
`|6|7|8|9|3|4|`
For comparison if the capacity would not be preserved the internal buffer would then result in `|1|2|5|6|7|8|9|3|4|`.

See Also:

`insert(iterator, value_type)`, `insert(iterator, size_type, value_type)`, `rinsert(iterator, value_type)`, `rinsert(iterator, size_type, value_type)`, `rinsert(iterator, InputIterator, InputIterator)`

Parameters:	`first` The beginning of the range to be inserted. `last` The end of the range to be inserted. `pos` An iterator specifying the position where the range will be inserted.
Requires:	`pos` is a valid iterator pointing to the `circular_buffer_space_optimized` or its end. Valid range `[first, last)` where `first` and `last` meet the requirements of an InputIterator.
Postconditions:	Elements from the range `[first + max[0, distance(first, last) - (pos - begin()) - reserve()], last)` will be inserted at the position `pos`. The number of `min[pos - begin(), max[0, distance(first, last) - reserve()]]` elements will be overwritten at the beginning of the `circular_buffer_space_optimized`. (See *Example* for the explanation.) The amount of allocated memory in the internal buffer may be predictively increased.
Throws:	An allocation error if memory is exhausted (`std::bad_alloc` if the standard allocator is used). Whatever `T::T(const T&)` throws or nothing if `T::T(T&&)` is noexcept.

26.
```
iterator rinsert(iterator pos, param_value_type item);
```

Insert an element before the specified position.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Exception Safety. Basic.

Iterator Invalidation. Invalidates all iterators pointing to the `circular_buffer_space_optimized` (except iterators equal to `end()`).

Complexity. Linear (in the size of the `circular_buffer_space_optimized`).

See Also:

`rinsert(iterator, size_type, value_type)`, `rinsert(iterator, InputIterator, InputIterator)`, `insert(iterator, value_type)`, `insert(iterator, size_type, value_type)`, `insert(iterator, InputIterator, InputIterator)`

Parameters:	`item` The element to be inserted. `pos` An iterator specifying the position before which the `item` will be inserted.
Requires:	`pos` is a valid iterator pointing to the `circular_buffer_space_optimized` or its end.
Postconditions:	The `item` will be inserted before the position `pos`. If the `circular_buffer_space_optimized` is full, the last element will be overwritten. If the `circular_buffer_space_optimized` is full and the `pos` points to `end()`, then the `item` will not be inserted. If the capacity is `0`, nothing will be inserted. The amount of allocated memory in the internal buffer may be predictively increased.
Returns:	Iterator to the inserted element or `end()` if the `item` is not inserted. (See the *Effect*.)
Throws:	An allocation error if memory is exhausted (`std::bad_alloc` if the standard allocator is used). Whatever `T::T(const T&)` throws. Whatever `T::operator = (const T&)` throws.

27.
```
iterator rinsert(iterator pos, rvalue_type item);
```

Insert an element before the specified position.

Exception Safety. Basic.

Iterator Invalidation. Invalidates all iterators pointing to the `circular_buffer_space_optimized` (except iterators equal to `end()`).

Complexity. Linear (in the size of the `circular_buffer_space_optimized`).

See Also:

`rinsert(iterator, size_type, value_type)`, `rinsert(iterator, InputIterator, InputIterator)`, `insert(iterator, value_type)`, `insert(iterator, size_type, value_type)`, `insert(iterator, InputIterator, InputIterator)`

Parameters:	`item` The element to be inserted. `pos` An iterator specifying the position before which the `item` will be inserted.
Requires:	`pos` is a valid iterator pointing to the `circular_buffer_space_optimized` or its end.
Postconditions:	The `item` will be inserted before the position `pos`. If the `circular_buffer_space_optimized` is full, the last element will be overwritten. If the `circular_buffer_space_optimized` is full and the `pos` points to `end()`, then the `item` will not be inserted. If the capacity is `0`, nothing will be inserted. The amount of allocated memory in the internal buffer may be predictively increased.
Returns:	Iterator to the inserted element or `end()` if the `item` is not inserted. (See the *Effect*.)
Throws:	An allocation error if memory is exhausted (`std::bad_alloc` if the standard allocator is used). Whatever `T::T(const T&)` throws or nothing if `T::T(T&&)` is noexcept.

28.
```
iterator rinsert(iterator pos);
```

Insert an element before the specified position.

Exception Safety. Basic.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Iterator Invalidation. Invalidates all iterators pointing to the `circular_buffer_space_optimized` (except iterators equal to `end()`).

Complexity. Linear (in the size of the `circular_buffer_space_optimized`).

See Also:

`rinsert(iterator, size_type, value_type)`, `rinsert(iterator, InputIterator, InputIterator)`, `insert(iterator, value_type)`, `insert(iterator, size_type, value_type)`, `insert(iterator, InputIterator, InputIterator)`

Parameters:	`pos` An iterator specifying the position before which the `item` will be inserted.
Requires:	`pos` is a valid iterator pointing to the `circular_buffer_space_optimized` or its end.
Postconditions:	The `item` will be inserted before the position `pos`. If the `circular_buffer_space_optimized` is full, the last element will be overwritten. If the `circular_buffer_space_optimized` is full and the `pos` points to `end()`, then the `item` will not be inserted. If the capacity is `0`, nothing will be inserted. The amount of allocated memory in the internal buffer may be predictively increased.
Returns:	Iterator to the inserted element or `end()` if the `item` is not inserted. (See the *Effect*.)
Throws:	An allocation error if memory is exhausted (`std::bad_alloc` if the standard allocator is used). Whatever `T::T()` throws. Whatever `T::T(const T&)` throws or nothing if `T::T(T&&)` is noexcept.

29.
```
void rinsert(iterator pos, size_type n, param_value_type item);
```

Insert `n` copies of the `item` before the specified position.

Exception Safety. Basic.

Iterator Invalidation. Invalidates all iterators pointing to the `circular_buffer_space_optimized` (except iterators equal to `end()`).

Complexity. Linear (in `min[capacity().capacity(), size() + n]`).

Example. Consider a `circular_buffer_space_optimized` with the capacity of 6 and the size of 4. Its internal buffer may look like the one below.
`|1|2|3|4| | |`
`p ___^`
After inserting 5 elements before the position `p`:
`rinsert(p, (size_t)5, 0);`
actually only 4 elements get inserted and elements `3` and `4` are overwritten. This is due to the fact the rinsert operation preserves the capacity. After insertion the internal buffer looks like this:
`|1|2|0|0|0|0|`
For comparison if the capacity would not be preserved the internal buffer would then result in `|1|2|0|0|0|0|0|3|4|`.

See Also:

`rinsert(iterator, value_type)`, `rinsert(iterator, InputIterator, InputIterator)`, `insert(iterator, value_type)`, `insert(iterator, size_type, value_type)`, `insert(iterator, InputIterator, InputIterator)`

Parameters:	`item`	The element whose copies will be inserted.
	`n`	The number of `item`s the to be inserted.
	`pos`	An iterator specifying the position where the `item`s will be inserted.
Requires:	`pos` is a valid iterator pointing to the `circular_buffer_space_optimized` or its end.	
Postconditions:	The number of `min[n, (end() - pos) + reserve()]` elements will be inserted before the position `pos`. The number of `min[end() - pos, max[0, n - reserve()]]` elements will be overwritten at the end of the `circular_buffer_space_optimized`. (See *Example* for the explanation.) The amount of allocated memory in the internal buffer may be predictively increased.	
Throws:	An allocation error if memory is exhausted (`std::bad_alloc` if the standard allocator is used). Whatever `T::T(const T&)` throws. Whatever `T::operator = (const T&)` throws.	

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

30.
```
template<typename InputIterator>
  void rinsert(iterator pos, InputIterator first, InputIterator last);
```

Insert the range `[first, last)` before the specified position.

Exception Safety. Basic.

Iterator Invalidation. Invalidates all iterators pointing to the `circular_buffer_space_optimized` (except iterators equal to `end()`).

Complexity. Linear (in `[size() + std::distance(first, last)]`; in `min[capacity().capacity(), size() + std::distance(first, last)]` if the `InputIterator` is a RandomAccessIterator).

Example. Consider a `circular_buffer_space_optimized` with the capacity of 6 and the size of 4. Its internal buffer may look like the one below.
`|1|2|3|4| | |`
`p` ___^
After inserting a range of elements before the position `p`:
`int array[] = { 5, 6, 7, 8, 9 };`
`insert(p, array, array + 5);`
actually only elements `5`, `6`, `7` and `8` from the specified range get inserted and elements `3` and `4` are overwritten. This is due to the fact the rinsert operation preserves the capacity. After insertion the internal buffer looks like this:
`|1|2|5|6|7|8|`
For comparison if the capacity would not be preserved the internal buffer would then result in `|1|2|5|6|7|8|9|3|4|`.

See Also:

`rinsert(iterator, value_type)`, `rinsert(iterator, size_type, value_type)`, `insert(iterator, value_type)`, `insert(iterator, size_type, value_type)`, `insert(iterator, InputIterator, InputIterator)`

Parameters:	`first`	The beginning of the range to be inserted.
	`last`	The end of the range to be inserted.
	`pos`	An iterator specifying the position where the range will be inserted.
Requires:	`pos` is a valid iterator pointing to the `circular_buffer_space_optimized` or its end. Valid range `[first, last)` where `first` and `last` meet the requirements of an InputIterator.	
Postconditions:	Elements from the range `[first, last - max[0, distance(first, last) - (end() - pos) - reserve()])` will be inserted before the position `pos`. The number of `min[end() - pos, max[0, distance(first, last) - reserve()]]` elements will be overwritten at the end of the `circular_buffer`. (See *Example* for the explanation.) The amount of allocated memory in the internal buffer may be predictively increased.	
Throws:	An allocation error if memory is exhausted (`std::bad_alloc` if the standard allocator is used). Whatever `T::T(const T&)` throws. Whatever `T::operator = (const T&)` throws.	

31.
```
iterator erase(iterator pos);
```

Remove an element at the specified position.

Exception Safety. Basic.

Iterator Invalidation. Invalidates all iterators pointing to the `circular_buffer_space_optimized` (except iterators equal to `end()`).

Complexity. Linear (in the size of the `circular_buffer_space_optimized`).

See Also:

`erase(iterator, iterator)`, `rerase(iterator)`, `rerase(iterator, iterator)`, `clear()`

Parameters:	`pos`	An iterator pointing at the element to be removed.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Requires: `pos` is a valid iterator pointing to the `circular_buffer_space_optimized` (but not an `end()`).
Postconditions: The element at the position `pos` is removed.
The amount of allocated memory in the internal buffer may be predictively decreased.
Returns: Iterator to the first element remaining beyond the removed element or `end()` if no such element exists.
Throws: An allocation error if memory is exhausted (`std::bad_alloc` if the standard allocator is used). Whatever `T::operator = (const T&)` throws or nothing if `T::operator = (T&&)` is noexcept.

32.

```
iterator erase(iterator first, iterator last);
```

Erase the range `[first, last)`.

Exception Safety. Basic.

Iterator Invalidation. Invalidates all iterators pointing to the `circular_buffer_space_optimized` (except iterators equal to `end()`).

Complexity. Linear (in the size of the `circular_buffer_space_optimized`).

See Also:

`erase(iterator)`, `rerase(iterator)`, `rerase(iterator, iterator)`, `clear()`
Parameters: `first` The beginning of the range to be removed.
`last` The end of the range to be removed.
Requires: Valid range `[first, last)`.
Postconditions: The elements from the range `[first, last)` are removed. (If `first == last` nothing is removed.)
The amount of allocated memory in the internal buffer may be predictively decreased.
Returns: Iterator to the first element remaining beyond the removed elements or `end()` if no such element exists.
Throws: An allocation error if memory is exhausted (`std::bad_alloc` if the standard allocator is used). Whatever `T::operator = (const T&)` throws or nothing if `T::operator = (T&&)` is noexcept.

33.

```
iterator rerase(iterator pos);
```

Remove an element at the specified position.

Exception Safety. Basic.

Iterator Invalidation. Invalidates all iterators pointing to the `circular_buffer_space_optimized` (except iterators equal to `end()`).

Complexity. Linear (in the size of the `circular_buffer_space_optimized`).

Note

Basically there is no difference between `erase(iterator)` and this method. It is implemented only for consistency with the base `circular_buffer`.

See Also:

`erase(iterator)`, `erase(iterator, iterator)`, `rerase(iterator, iterator)`, `clear()`
Parameters: `pos` An iterator pointing at the element to be removed.
Requires: `pos` is a valid iterator pointing to the `circular_buffer_space_optimized` (but not an `end()`).
The amount of allocated memory in the internal buffer may be predictively decreased.
Postconditions: The element at the position `pos` is removed.
Returns: Iterator to the first element remaining in front of the removed element or `begin()` if no such element exists.
Throws: An allocation error if memory is exhausted (`std::bad_alloc` if the standard allocator is used). Whatever `T::operator = (const T&)` throws or nothing if `T::operator = (T&&)` is noexcept.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

34.
```
iterator rerase(iterator first, iterator last);
```

Erase the range `[first, last)`.

Exception Safety. Basic.

Iterator Invalidation. Invalidates all iterators pointing to the `circular_buffer_space_optimized` (except iterators equal to `end()`).

Complexity. Linear (in the size of the `circular_buffer_space_optimized`).

> ### Note
>
> Basically there is no difference between `erase(iterator, iterator)` and this method. It is implemented only for consistency with the base <`circular_buffer`.

See Also:

`erase(iterator)`, `erase(iterator, iterator)`, `rerase(iterator)`, `clear()`

Parameters:	`first`	The beginning of the range to be removed.
	`last`	The end of the range to be removed.
Requires:	Valid range `[first, last)`.	
Postconditions:	The elements from the range `[first, last)` are removed. (If `first == last` nothing is removed.) The amount of allocated memory in the internal buffer may be predictively decreased.	
Returns:	Iterator to the first element remaining in front of the removed elements or `begin()` if no such element exists.	
Throws:	An allocation error if memory is exhausted (`std::bad_alloc` if the standard allocator is used). Whatever `T::operator = (const T&)` throws or nothing if `T::operator = (T&&)` is noexcept.	

35.
```
void clear();
```

Remove all stored elements from the space optimized circular buffer.

Exception Safety. Basic.

Iterator Invalidation. Invalidates all iterators pointing to the `circular_buffer_space_optimized` (except iterators equal to `end()`).

Complexity. Linear (in the size of the `circular_buffer_space_optimized`).

See Also:

`~circular_buffer_space_optimized()`, `erase(iterator)`, `erase(iterator, iterator)`, `rerase(iterator)`, `rerase(iterator, iterator)`

Postconditions:	`size() == 0` The amount of allocated memory in the internal buffer may be predictively decreased.
Throws:	An allocation error if memory is exhausted (`std::bad_alloc` if the standard allocator is used).

`circular_buffer_space_optimized` private member functions

1.
```
void adjust_min_capacity();
```

Adjust the amount of allocated memory.

2.
```
size_type ensure_reserve(size_type new_capacity, size_type buffer_size) const;
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Ensure the reserve for possible growth up.

3.
```
void check_low_capacity(size_type n = 1);
```

Check for low capacity.

4.
```
void check_high_capacity();
```

Check for high capacity.

5.
```
void reduce_capacity(const true_type &);
```

Specialized method for reducing the capacity.

6.
```
void reduce_capacity(const false_type &);
```

Specialized method for reducing the capacity.

7.
```
template<typename IntegralType>
  void insert(const iterator & pos, IntegralType n, IntegralType item,
              const true_type &);
```

Specialized insert method.

8.
```
template<typename Iterator>
  void insert(const iterator & pos, Iterator first, Iterator last,
              const false_type &);
```

Specialized insert method.

9.
```
template<typename IntegralType>
  void rinsert(const iterator & pos, IntegralType n, IntegralType item,
               const true_type &);
```

Specialized rinsert method.

10.
```
template<typename Iterator>
  void rinsert(const iterator & pos, Iterator first, Iterator last,
               const false_type &);
```

Specialized rinsert method.

`circular_buffer_space_optimized` private static functions

1.
```
static size_type
init_capacity(const capacity_type & capacity_ctrl, size_type n);
```

Determine the initial capacity.

2.
```
template<typename IntegralType>
  static size_type
  init_capacity(const capacity_type & capacity_ctrl, IntegralType n,
                IntegralType, const true_type &);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Specialized method for determining the initial capacity.

3.
```
template<typename Iterator>
  static size_type
  init_capacity(const capacity_type & capacity_ctrl, Iterator first,
                Iterator last, const false_type &);
```

Specialized method for determining the initial capacity.

4.
```
template<typename InputIterator>
  static size_type
  init_capacity(const capacity_type & capacity_ctrl, InputIterator,
                InputIterator, const std::input_iterator_tag &);
```

Specialized method for determining the initial capacity.

5.
```
template<typename ForwardIterator>
  static size_type
  init_capacity(const capacity_type & capacity_ctrl, ForwardIterator first,
                ForwardIterator last, const std::forward_iterator_tag &);
```

Specialized method for determining the initial capacity.

Index

A

add
    Class template circular_buffer, 22, 56
adjust_min_capacity
    Class template circular_buffer_space_optimized, 66, 87
allocate
    Class template circular_buffer, 22, 56
allocator_type
    Class template circular_buffer, 22
    Class template circular_buffer_space_optimized, 66
array_one
    Class template circular_buffer, 22, 35-37
array_range
    Class template circular_buffer, 22, 26
    Class template circular_buffer_space_optimized, 66
array_two
    Class template circular_buffer, 22, 35-37
assign
    Class template circular_buffer, 22, 42-43, 58-59
    Class template circular_buffer_space_optimized, 66, 74-76
assign_n
    Class template circular_buffer, 22, 59
at
    Class template circular_buffer, 22, 33-34
    Implementation , 6

B

back
    Class template circular_buffer, 22, 34-35, 44-45
    Class template circular_buffer_space_optimized, 77-78

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

begin
    Class template circular_buffer, 22, 30-32, 38, 47-48, 53-54
    Class template circular_buffer_space_optimized, 80-81, 86-87
BOOST_CB_DISABLE_DEBUG
    Class template circular_buffer_space_optimized, 76
    Implementation , 6
bounded_buffer
    More Examples, 10

C

capacity_control
    Class template circular_buffer_space_optimized, 68
capacity_type
    Class template circular_buffer, 22, 26
    Class template circular_buffer_space_optimized, 66, 68
check_high_capacity
    Class template circular_buffer_space_optimized, 66, 88
check_low_capacity
    Class template circular_buffer_space_optimized, 66, 88
circular_buffer_space_optimized
    Class template circular_buffer_space_optimized, 66
Class template circular_buffer
    add, 22, 56
    allocate, 22, 56
    allocator_type, 22
    array_one, 22, 35-37
    array_range, 22, 26
    array_two, 22, 35-37
    assign, 22, 42-43, 58-59
    assign_n, 22, 59
    at, 22, 33-34
    back, 22, 34-35, 44-45
    begin, 22, 30-32, 38, 47-48, 53-54
    capacity_type, 22, 26
    clear, 22, 55
    construct_or_replace, 22, 56
    const_array_range, 22, 26
    const_iterator, 22
    const_pointer, 22
    const_reference, 22
    const_reverse_iterator, 22
    deallocate, 22, 56
    decrement, 22, 56
    destroy, 22, 57
    destroy_content, 22, 57
    destroy_if_constructed, 22, 57
    destroy_item, 22, 57
    difference_type, 22, 26
    end, 22, 29-31, 37-38, 40-43, 47-53, 55
    erase, 22, 52-54
    erase_begin, 22, 54, 61
    erase_end, 22, 54, 61
    front, 22, 34-35, 45-46
    initialize, 22, 57-58
    initialize_buffer, 22, 57
    insert, 22, 27, 47-49, 59-60
    insert_impl, 22, 55

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

  insert_item, 22, 59
  insert_n, 22, 60
  iterator, 22
  Linear, 27-29, 37-38, 40-43, 47-53
  linearize, 22, 35, 37
  param_value_type, 22
  pointer, 22
  pop_back, 22, 46
  pop_front, 22, 46
  push_back, 22, 27, 44-45
  push_back_impl, 22, 55
  push_front, 22, 45-46
  push_front_impl, 22, 55
  reference, 22
  replace, 22, 56
  rerase, 22, 53-54
  reset, 22, 58
  reverse_iterator, 22
  rinsert, 22, 49-51, 60
  rinsert_impl, 22, 55
  rinsert_n, 22, 60
  rotate, 22, 38
  rset_capacity, 22, 41
  rvalue_type, 22, 26
  set_capacity, 22, 27, 40
  size_type, 22, 26
  swap_allocator, 22, 58
  this_type, 22
  value_type, 22
  write, 35
Class template circular_buffer_space_optimized
  adjust_min_capacity, 66, 87
  allocator_type, 66
  array_range, 66
  assign, 66, 74-76
  back, 77-78
  begin, 80-81, 86-87
  BOOST_CB_DISABLE_DEBUG, 76
  capacity_control, 68
  capacity_type, 66, 68
  check_high_capacity, 66, 88
  check_low_capacity, 66, 88
  circular_buffer_space_optimized, 66
  clear, 66, 87
  const_array_range, 66
  const_iterator, 66
  const_pointer, 66
  const_reference, 66
  const_reverse_iterator, 66
  difference_type, 66
  end, 71-87
  erase, 66, 85-87
  front, 78-79
  init_capacity, 66, 88-89
  insert, 66, 80-82, 88
  iterator, 66
  Linear, 69-87
  param_value_type, 66

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

  - pointer, 66
  - pop_back, 66, 79
  - pop_front, 66, 79
  - push_back, 66, 77
  - push_front, 66, 78-79
  - reduce_capacity, 66, 88
  - reference, 66
  - rerase, 66, 86-87
  - reverse_iterator, 66
  - rinsert, 66, 82-85, 88
  - rset_capacity, 66, 73
  - rvalue_type, 66
  - set_capacity, 66, 72
  - size_type, 66
  - value_type, 66
- clear
  - Class template circular_buffer, 22, 55
  - Class template circular_buffer_space_optimized, 66, 87
  - Release Notes, 17
- construct_or_replace
  - Class template circular_buffer, 22, 56
- const_array_range
  - Class template circular_buffer, 22, 26
  - Class template circular_buffer_space_optimized, 66
- const_iterator
  - Class template circular_buffer, 22
  - Class template circular_buffer_space_optimized, 66
- const_pointer
  - Class template circular_buffer, 22
  - Class template circular_buffer_space_optimized, 66
- const_reference
  - Class template circular_buffer, 22
  - Class template circular_buffer_space_optimized, 66
- const_reverse_iterator
  - Class template circular_buffer, 22
  - Class template circular_buffer_space_optimized, 66
- container_type
  - More Examples, 10

D

- deallocate
  - Class template circular_buffer, 22, 56
- decrement
  - Class template circular_buffer, 22, 56
- destroy
  - Class template circular_buffer, 22, 57
- destroy_content
  - Class template circular_buffer, 22, 57
- destroy_if_constructed
  - Class template circular_buffer, 22, 57
- destroy_item
  - Class template circular_buffer, 22, 57
- difference_type
  - Class template circular_buffer, 22, 26
  - Class template circular_buffer_space_optimized, 66

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

E

end
    Class template circular_buffer, 22, 29-31, 37-38, 40-43, 47-53, 55
    Class template circular_buffer_space_optimized, 71-87
erase
    Class template circular_buffer, 22, 52-54
    Class template circular_buffer_space_optimized, 66, 85-87
erase_begin
    Class template circular_buffer, 22, 54, 61
    Release Notes, 17
erase_end
    Class template circular_buffer, 22, 54, 61
    Release Notes, 17

F

front
    Class template circular_buffer, 22, 34-35, 45-46
    Class template circular_buffer_space_optimized, 78-79
    Implementation , 6
Function template operator!=
    Linear, 62
Function template operator<
    Linear, 62
Function template operator<=
    Linear, 63
Function template operator==
    Linear, 61
Function template operator>
    Linear, 62
Function template operator>=
    Linear, 64

I

Implementation
    at, 6
    BOOST_CB_DISABLE_DEBUG, 6
    front, 6
    main, 6
    pop_back, 6
initialize
    Class template circular_buffer, 22, 57-58
initialize_buffer
    Class template circular_buffer, 22, 57
init_capacity
    Class template circular_buffer_space_optimized, 66, 88-89
insert
    Class template circular_buffer, 22, 27, 47-49, 59-60
    Class template circular_buffer_space_optimized, 66, 80-82, 88
insert_impl
    Class template circular_buffer, 22, 55
insert_item
    Class template circular_buffer, 22, 59
insert_n
    Class template circular_buffer, 22, 60
iterator
    Class template circular_buffer, 22
    Class template circular_buffer_space_optimized, 66

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

L

Linear
  Class template circular_buffer, 27-29, 37-38, 40-43, 47-53
  Class template circular_buffer_space_optimized, 69-87
  Function template operator!=, 62
  Function template operator<, 62
  Function template operator<=, 63
  Function template operator==, 61
  Function template operator>, 62
  Function template operator>=, 64
linearize
  Class template circular_buffer, 22, 35, 37
  Trac Tickets, 16
lock
  More Examples, 10

M

main
  Implementation , 6
  More Examples, 10
More Examples
  bounded_buffer, 10
  container_type, 10
  lock, 10
  main, 10
  param_type, 10
  pop_back, 10
  push_front, 10
  size_type, 10
  value_type, 10

P

param_type
  More Examples, 10
param_value_type
  Class template circular_buffer, 22
  Class template circular_buffer_space_optimized, 66
pointer
  Class template circular_buffer, 22
  Class template circular_buffer_space_optimized, 66
pop_back
  Class template circular_buffer, 22, 46
  Class template circular_buffer_space_optimized, 66, 79
  Implementation , 6
  More Examples, 10
pop_front
  Class template circular_buffer, 22, 46
  Class template circular_buffer_space_optimized, 66, 79
push_back
  Class template circular_buffer, 22, 27, 44-45
  Class template circular_buffer_space_optimized, 66, 77
push_back_impl
  Class template circular_buffer, 22, 55
push_front
  Class template circular_buffer, 22, 45-46
  Class template circular_buffer_space_optimized, 66, 78-79
  More Examples, 10

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

push_front_impl
    Class template circular_buffer, 22, 55

R
reduce_capacity
    Class template circular_buffer_space_optimized, 66, 88
reference
    Class template circular_buffer, 22
    Class template circular_buffer_space_optimized, 66
Release Notes
    clear, 17
    erase_begin, 17
    erase_end, 17
    rotate, 17
replace
    Class template circular_buffer, 22, 56
rerase
    Class template circular_buffer, 22, 53-54
    Class template circular_buffer_space_optimized, 66, 86-87
reset
    Class template circular_buffer, 22, 58
reverse_iterator
    Class template circular_buffer, 22
    Class template circular_buffer_space_optimized, 66
rinsert
    Class template circular_buffer, 22, 49-51, 60
    Class template circular_buffer_space_optimized, 66, 82-85, 88
rinsert_impl
    Class template circular_buffer, 22, 55
rinsert_n
    Class template circular_buffer, 22, 60
rotate
    Class template circular_buffer, 22, 38
    Release Notes, 17
rset_capacity
    Class template circular_buffer, 22, 41
    Class template circular_buffer_space_optimized, 66, 73
rvalue_type
    Class template circular_buffer, 22, 26
    Class template circular_buffer_space_optimized, 66

S
set_capacity
    Class template circular_buffer, 22, 27, 40
    Class template circular_buffer_space_optimized, 66, 72
size_type
    Class template circular_buffer, 22, 26
    Class template circular_buffer_space_optimized, 66
    More Examples, 10
swap_allocator
    Class template circular_buffer, 22, 58

T
this_type
    Class template circular_buffer, 22
Trac Tickets
    linearize, 16

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

V

value_type
- Class template circular_buffer, 22
- Class template circular_buffer_space_optimized, 66
- More Examples, 10

W

write
- Class template circular_buffer, 35

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/