Templated Circular Buffer Container

circular_buffer<T, Alloc>

Contents

[Description](#)
[Introductory Example](#)
[Synopsis](#)
[Rationale](#)

- [Thread-Safety](#)
- [Overwrite Operation](#)
- [Writing to a Full Buffer](#)
- [Reading/Removing from an Empty Buffer](#)
- [Iterator Invalidation](#)

[Caveats](#)
[Debug Support](#)
[Compatibility with Interprocess library](#)
[More Examples](#)
[Header Files](#)
[Modelled Concepts](#)
[Template Parameters](#)
[Public Types](#)
[Constructors and Destructor](#)
[Public Member Functions](#)
[Standalone Functions](#)
[Notes](#)
[See also](#)
[Acknowledgements](#)
[Release Notes](#)

Description

In general the term *circular 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. (Also see the Figure.)

The `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 either at the

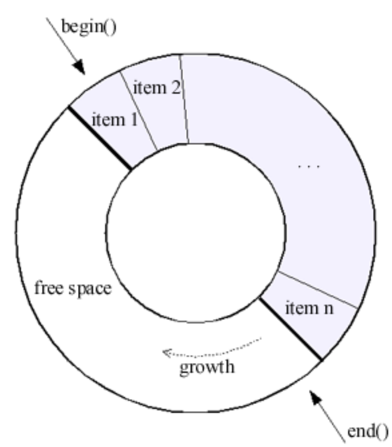

Figure: The circular buffer (for someone known beginning or end of the buffer (depending on what insert operation is used) to be overwritten.

as *ring* or *cyclic buffer*).

The circular_buffer only allocates memory when created, when the capacity is adjusted explicitly, or as necessary to accommodate resizing or assign operations. On the other hand, there is also a circular_buffer_space_optimized available. It is an adaptor of the circular_buffer which does not allocate memory at once when created, rather it allocates memory as needed.

Introductory Example

A brief example using the circular_buffer:

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

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

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

   // Insert some 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, 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.

   int d = cb[0];  // d == 4


   return 0;
}
```

Synopsis

```
namespace boost {

template <class T, class Alloc>
class circular_buffer
{
public:
```

```
typedef typename Alloc::value_type value_type;
typedef typename Alloc::pointer pointer;
typedef typename Alloc::const_pointer const_pointer;
typedef typename Alloc::reference reference;
typedef typename Alloc::const_reference const_reference;
typedef typename Alloc::difference_type difference_type;
typedef typename Alloc::size_type size_type;
typedef Alloc allocator_type;
typedef implementation-defined const_iterator;
typedef implementation-defined iterator;
typedef boost::reverse_iterator<const_iterator> const_reverse_iterator;
typedef boost::reverse_iterator<iterator> reverse_iterator;
typedef std::pair<pointer, size_type> array_range;
typedef std::pair<const_pointer, size_type> const_array_range;
typedef size_type capacity_type;

explicit circular_buffer(const allocator_type& alloc = allocator_type());
explicit circular_buffer(capacity_type buffer_capacity, const allocator_type& alloc =
circular_buffer(size_type n, const_reference item, const allocator_type& alloc = allo
circular_buffer(capacity_type buffer_capacity, size_type n, const_reference item, cor
circular_buffer(const circular_buffer<T, Alloc>& cb);
template <class InputIterator>
   circular_buffer(InputIterator first, InputIterator last, const allocator_type& all
template <class InputIterator>
   circular_buffer(capacity_type buffer_capacity, InputIterator first, InputIterator
~circular_buffer();

allocator_type get_allocator() const;
allocator_type& get_allocator();
iterator begin();
iterator end();
const_iterator begin() const;
const_iterator end() const;
reverse_iterator rbegin();
reverse_iterator rend();
const_reverse_iterator rbegin() const;
const_reverse_iterator rend() const;
reference operator[](size_type index);
const_reference operator[](size_type index) const;
reference at(size_type index);
const_reference at(size_type index) 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;
void rotate(const_iterator new_begin);
size_type size() const;
size_type max_size() const;
bool empty() const;
bool full() const;
size_type reserve() const;
capacity_type capacity() const;
void set_capacity(capacity_type new_capacity);
void resize(size_type new_size, const_reference item = value_type());
void rset_capacity(capacity_type new_capacity);
void rresize(size_type new_size, const_reference item = value_type());
circular_buffer<T, Alloc>& operator=(const circular_buffer<T, Alloc>& cb);
```

```
   void assign(size_type n, const_reference item);
   void assign(capacity_type buffer_capacity, size_type n, const_reference item);
   template <class InputIterator>
      void assign(InputIterator first, InputIterator last);
   template <class InputIterator>
      void assign(capacity_type buffer_capacity, InputIterator first, InputIterator last
   void swap(circular_buffer<T, Alloc>& cb);
   void push_back(const_reference item = value_type());
   void push_front(const_reference item = value_type());
   void pop_back();
   void pop_front();
   iterator insert(iterator pos, const_reference item = value_type());
   void insert(iterator pos, size_type n, const_reference item);
   template <class InputIterator>
      void insert(iterator pos, InputIterator first, InputIterator last);
   iterator rinsert(iterator pos, const_reference item = value_type());
   void rinsert(iterator pos, size_type n, const_reference item);
   template <class InputIterator>
      void rinsert(iterator pos, InputIterator first, InputIterator last);
   iterator erase(iterator pos);
   iterator erase(iterator first, iterator last);
   iterator rerase(iterator pos);
   iterator rerase(iterator first, iterator last);
   void erase_begin(size_type n);
   void erase_end(size_type n);
   void clear();
};

template <class T, class Alloc>
   bool operator==(const circular_buffer<T, Alloc>& lhs, const circular_buffer<T, Alloc>
template <class T, class Alloc>
   bool operator<(const circular_buffer<T, Alloc>& lhs, const circular_buffer<T, Alloc>&
template <class T, class Alloc>
   bool operator!=(const circular_buffer<T, Alloc>& lhs, const circular_buffer<T, Alloc>
template <class T, class Alloc>
   bool operator>(const circular_buffer<T, Alloc>& lhs, const circular_buffer<T, Alloc>&
template <class T, class Alloc>
   bool operator<=(const circular_buffer<T, Alloc>& lhs, const circular_buffer<T, Alloc>
template <class T, class Alloc>
   bool operator>=(const circular_buffer<T, Alloc>& lhs, const circular_buffer<T, Alloc>
template <class T, class Alloc>
   void swap(circular_buffer<T, Alloc>& lhs, circular_buffer<T, Alloc>& rhs);

} // namespace boost
```

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:

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

In order to achieve maximum efficiency, the `circular_buffer` stores its elements in a *contiguous region of memory*, which then enables:

1. Use of fixed memory and no implicit or unexpected memory allocation.
2. Fast constant-time insertion and removal of elements from the front and back.
3. Fast constant-time random access of elements.
4. 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).
- A kind of cache storing a specified number of last inserted elements.
- Efficient fixed capacity FIFO (First In, First Out) or LIFO (Last In, First Out) queue which removes the oldest (inserted as first) elements when full.

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** thread-safe. The thread-safety is guarantied 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.)

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 with the case if a data source produces more data than can fit in the fixed-sized buffer:

1. Inform the data source to wait until there is room in the buffer (e.g. by throwing an overflow exception).

2. If the oldest data is the most important, ignore new data from the source until there is room in the buffer again.
3. If the latest data is the most important, write over the oldest data.
4. 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, in 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:

1. It would introduce performance overhead.
2. 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. 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 an 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

#include <boost/circular_buffer.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.

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 [1]. (Any container of `std::auto_ptr` is considered particularly hazardous. [2])

While internals of a `circular_buffer` are circular, iterators are **not**. Iterators of a `circular_buffer` are only valid for the range `[begin(), end()]`. E.g. iterators `(begin() - 1)` and `(end() + 1)` are 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 (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. This can help the programmer when debugging the code to recognize the initialized memory from the uninitialized. For details refer the source code.

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 `BOOST_CB_DISABLE_DEBUG` macro.

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.

More Examples

The following example includes various usage of the circular_buffer.

```
#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);

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

   // assertions
   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
   int sum = std::accumulate(cb.begin(), cb.end(), 0);

   // assertions
   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(sum == 9);
   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 not 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.

Bounded Buffer Example

The bounded buffer is normally used in a producer-consumer mode when 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, that consumers do not try to remove items when the container is empty, and that each produced item is consumed by exactly one consumer.

The example below 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/progress.hpp>
#include <boost/bind.hpp>

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(boost::call_traits<value_type>::param_type item) {
      // param_type represents the "best" way to pass a parameter of type value_type

      boost::mutex::scoped_lock lock(m_mutex);
      m_not_full.wait(lock, boost::bind(&bounded_buffer<value_type>::is_not_full, thi
      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, t
      *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(); }

   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 Threads](#) 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 `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 `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.)

Header Files

The `circular_buffer` is defined in the file boost/circular_buffer.hpp. There is also a forward declaration for the `circular_buffer` in the header file boost/circular_buffer_fwd.hpp.

Modelled Concepts

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

Template Parameters

Parameter	Description	Default
`T`	The type of the elements stored in the `circular_buffer`. **Type Requirements:** 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.	
`Alloc`	The allocator type used for all internal memory management. **Type Requirements:** The `Alloc` has to meet the allocator requirements imposed by STL.	`std::allocator<T>`

Public Types

Type	Description
value_type	The type of elements stored in the circular_buffer.
pointer	A pointer to an element.
const_pointer	A const pointer to the element.
reference	A reference to an element.
const_reference	A const reference to an element.
difference_type	The distance type. (A signed integral type used to represent the distance between two iterators.)
size_type	The size type. (An unsigned integral type that can represent any non-negative value of the container's distance type.)
allocator_type	The type of an allocator used in the circular_buffer.
const_iterator	A const (random access) iterator used to iterate through the circular_buffer.
iterator	A (random access) iterator used to iterate through the circular_buffer.
const_reverse_iterator	A const iterator used to iterate backwards through a circular_buffer.
reverse_iterator	An iterator used to iterate backwards through a circular_buffer.
array_range	An 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.)
const_array_range	A range of a const array. (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.)
capacity_type	The capacity type. (Same as size_type - defined for consistency with the circular_buffer_space_optimized.)

Constructors and Destructor

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

Create an empty circular_buffer with zero capacity.

Effect:
capacity() == 0 && size() == 0

Parameter(s):

alloc
The allocator.

Throws:
An allocation error if memory is exhausted (std::bad_alloc if the standard allocator is used).

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)

---

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

Create an empty `circular_buffer` with the specified capacity.

Effect:
`capacity() == buffer_capacity && size() == 0`

Parameter(s):

`buffer_capacity`
The maximum number of elements which can be stored in the `circular_buffer`.

`alloc`
The allocator.

Throws:
An allocation error if memory is exhausted (`std::bad_alloc` if the standard allocator is used).

Complexity:
Constant.

---

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

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

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

Parameter(s):

`n`
The number of elements the created `circular_buffer` will be filled with.

`item`
The element the created `circular_buffer` will be filled with.

`alloc`
The allocator.

Throws:

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

Complexity:
Linear (in the `n`).

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

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

Precondition:
`buffer_capacity >= n`

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

Parameter(s):

`buffer_capacity`
The capacity of the created `circular_buffer`.

`n`
The number of elements the created `circular_buffer` will be filled with.

`item`
The element the created `circular_buffer` will be filled with.

`alloc`
The allocator.

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

Complexity:
Linear (in the `n`).

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

The copy constructor.

Creates a copy of the specified `circular_buffer`.

Effect:
`*this == cb`

Parameter(s):

`cb`
The `circular_buffer` to 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.

Complexity:

Linear (in the size of `cb`).

---

template <class 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.

Precondition:

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

Effect:

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

Parameter(s):

`first`
The beginning of the range to be copied.

`last`
The end of the range to be copied.

`alloc`
The allocator.

Throws:

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

Complexity:

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

---

template <class 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.

Precondition:

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

Effect:

`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.

Parameter(s):

`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.

alloc
The allocator.

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

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

`~circular_buffer();`

The destructor.

Destroys the `circular_buffer`.

Throws:
Nothing.

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()`

Public Member Functions

`allocator_type get_allocator() const;`

Get the allocator.

Returns:
The allocator.

Throws:
Nothing.

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.

allocator_type& get_allocator();

Get the allocator reference.

Returns:
A reference to the allocator.

Throws:
Nothing.

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

---

iterator begin();

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

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.

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()

---

iterator end();

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

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.

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()

const_iterator begin() const;

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

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.

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

const_iterator end() const;

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

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.

Exception Safety:
No-throw.

Iterator Invalidation:
Does not invalidate any iterators.

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

See Also:

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

reverse_iterator rbegin();

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

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.

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()

reverse_iterator rend();

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

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.

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()

const_reverse_iterator rbegin() const;

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

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.

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`

`const_reverse_iterator rend() const;`

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

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.

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`

`reference operator[](size_type index);`

Get the element at the `index` position.

Precondition:
`0 <= index && index < size()`

Parameter(s):

`index`
The position of the element.

Returns:
A reference to the element at the `index` position.

Throws:
Nothing.

Exception Safety:
No-throw.

Iterator Invalidation:
Does not invalidate any iterators.

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

See Also:
`at()`

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

Get the element at the `index` position.

Precondition:
`0 <= index && index < size()`

Parameter(s):

`index`
The position of the element.

Returns:
A const reference to the element at the `index` position.

Throws:
Nothing.

Exception Safety:
No-throw.

Iterator Invalidation:
Does not invalidate any iterators.

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

See Also:
`at() const`

`reference` at(`size_type` index);

Get the element at the `index` position.

Parameter(s):

`index`
The position of the element.

Returns:
A reference to the element at the `index` position.

Throws:
`std::out_of_range` when the `index` is invalid (when `index >= size()`).

Exception Safety:
Strong.

Iterator Invalidation:
Does not invalidate any iterators.

Complexity:

Constant (in the size of the circular_buffer).

See Also:
operator[]

const_reference at(size_type index) const;

Get the element at the index position.

Parameter(s):

index
The position of the element.

Returns:
A const reference to the element at the index position.

Throws:
std::out_of_range when the index is invalid (when index >= size()).

Exception Safety:
Strong.

Iterator Invalidation:
Does not invalidate any iterators.

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

See Also:
operator[] const

reference front();

Get the first element.

Precondition:
!empty()

Returns:
A reference to the first element of the circular_buffer.

Throws:
Nothing.

Exception Safety:
No-throw.

Iterator Invalidation:
Does not invalidate any iterators.

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

See Also:
back()

reference back();

Get the last element.

Precondition:
`!empty()`

Returns:
A reference to the last element of the `circular_buffer`.

Throws:
Nothing.

Exception Safety:
No-throw.

Iterator Invalidation:
Does not invalidate any iterators.

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

See Also:
`front()`

`const_reference` `front() const;`

Get the first element.

Precondition:
`!empty()`

Returns:
A const reference to the first element of the `circular_buffer`.

Throws:
Nothing.

Exception Safety:
No-throw.

Iterator Invalidation:
Does not invalidate any iterators.

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

See Also:
`back() const`

`const_reference` `back() const;`

Get the last element.

Precondition:
`!empty()`

Returns:
A const reference to the last element of the `circular_buffer`.

Throws:
Nothing.

Exception Safety:

No-throw.

Iterator Invalidation:
Does not invalidate any iterators.

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

See Also:
[front() const]

[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();
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.

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.

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()`

`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.

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.

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_array_range` array_one() const;

Get the first continuous array of the internal buffer.

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

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.

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.

[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.

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.

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]()

[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.

Effect:
`&(*this)[0] < &(*this)[1] < ... < &(*this)[size() - 1]`

Returns:
A pointer to the beginning of the array or `0` if empty.

Throws:
Whatever `T::T(const T&)` throws.
Whatever `T::operator = (const T&)` throws.

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)`

`bool is_linearized() const;`

Is the `circular_buffer` linearized?

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.

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()`

`void rotate(const_iterator new_begin);`

Rotate elements in the `circular_buffer`.

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

Precondition:
`new_begin` is a valid iterator pointing to the `circular_buffer` **except** its end.

Effect:
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]`

Parameter(s):

`new_begin`
The new beginning.

Throws:
Whatever `T::T(const T&)` throws.
Whatever `T::operator = (const T&)` throws.

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]

[size_type] size() const;

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

Returns:
The number of elements stored in the `circular_buffer`.

Throws:
Nothing.

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)]

[size_type] max_size() const;

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

Returns:
The maximum size/capacity the `circular_buffer` can be set to.

Throws:

Nothing.

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()

`bool empty() const;`

Is the `circular_buffer` empty?

Returns:
`true` if there are no elements stored in the `circular_buffer`; `false` otherwise.

Throws:
Nothing.

Exception Safety:
No-throw.

Iterator Invalidation:
Does not invalidate any iterators.

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

See Also:
full()

`bool full() const;`

Is the `circular_buffer` full?

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

Throws:
Nothing.

Exception Safety:
No-throw.

Iterator Invalidation:
Does not invalidate any iterators.

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

See Also:
empty()

`size_type reserve() const;`

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

Returns:
capacity() - size()

Throws:
Nothing.

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()

capacity_type capacity() const;

Get the capacity of the circular_buffer.

Returns:
The maximum number of elements which can be stored in the circular_buffer.

Throws:
Nothing.

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)

void set_capacity(capacity_type new_capacity);

Change the capacity of the circular_buffer.

Effect:
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.

Parameter(s):

new_capacity
The 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.

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)

---

void resize(size_type new_size, const_reference item = value_type());

Change the size of the circular_buffer.

Effect:

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.)

Parameter(s):

new_size
: The new size.

item
: The element the circular_buffer will be filled with in order to gain the requested size. (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.

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)

void rset_capacity(capacity_type new_capacity);

Change the capacity of the `circular_buffer`.

Effect:

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`.

Parameter(s):

`new_capacity`

The 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.

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)

void rresize(size_type new_size, const_reference item = value_type());

Change the size of the `circular_buffer`.

Effect:

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.)

Parameter(s):

`new_size`

The new size.

`item`

The element the `circular_buffer` will be filled with in order to gain the requested size. (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.

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:
[resize(size_type, const_reference)], [rset_capacity(capacity_type)]

`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`.

Effect:
`*this == cb`

Parameter(s):

`cb`
The `circular_buffer` to 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.

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)]

`void assign(size_type n, const_reference 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`.

Effect:

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

Parameter(s):

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

`item`
: The element the `circular_buffer` will be filled with.

Throws:

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

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)`

`void assign(capacity_type buffer_capacity, size_type n, const_reference 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`.

Precondition:

`capacity >= n`

Effect:

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

Parameter(s):

`buffer_capacity`
: The new capacity.

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

`item`
: The element the `circular_buffer` will be filled with.

Throws:

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

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

```
template <class 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.

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

Effect:
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)

Parameter(s):

first
The beginning of the range to be copied.

last
The end of the range to 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.

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)

template <class 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.

Precondition:

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

Effect:

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.

Parameter(s):

buffer_capacity
: The new capacity.

first
: The beginning of the range to be copied.

last
: The end of the range to 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.

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)

void swap(circular_buffer<T,Alloc>& cb);

Swap the contents of two circular_buffers.

Effect:
this contains elements of cb and vice versa; the capacity of this equals to the capacity of cb and vice versa.

Parameter(s):

cb
The circular_buffer whose content will be swapped.

Throws:
Nothing.

Exception Safety:
No-throw.

Iterator Invalidation:
Invalidates all iterators of both circular_buffers. (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>&)

void push_back(const_reference item = value_type());

Insert a new element at the end of the circular_buffer.

Effect:
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.

Parameter(s):

item
The element to be inserted.

Throws:
Whatever T::T(const T&) throws.
Whatever T::operator = (const T&) throws.

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()

void push_front(const_reference item = value_type());

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

Effect:
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.

Parameter(s):

`item`
The element to be inserted.

Throws:
Whatever `T::T(const T&)` throws.
Whatever `T::operator = (const T&)` throws.

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()`

void pop_back();

Remove the last element from the `circular_buffer`.

Precondition:
`!empty()`

Effect:
The last element is removed from the `circular_buffer`.

Throws:
Nothing.

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)`

void pop_front();

Remove the first element from the `circular_buffer`.

Precondition:
`!empty()`

Effect:
The first element is removed from the `circular_buffer`.

Throws:
Nothing.

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)

iterator insert(iterator pos, const_reference item = value_type());

Insert an element at the specified position.

Precondition:
`pos` is a valid iterator pointing to the `circular_buffer` or its end.

Effect:
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.

Parameter(s):

`pos`
An iterator specifying the position where the `item` will be inserted.

`item`
The element to 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.

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)

void insert(iterator pos, size_type n, const_reference item);

Insert n copies of the item at the specified position.

Precondition:

pos is a valid iterator pointing to the circular_buffer or its end.

Effect:

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.
(See *Example* for the explanation.)

Parameter(s):

pos
: An iterator specifying the position where the items will be inserted.

n
: The number of items the to be inserted.

item
: The element whose copies will be inserted.

Throws:

Whatever T::T(const T&) throws.
Whatever T::operator = (const T&) throws.

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)

template <class InputIterator>
void insert(iterator pos, InputIterator first, InputIterator last);

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

Precondition:

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.

Effect:

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.)

Parameter(s):

pos
: An iterator specifying the position where the range will be inserted.

first
: The beginning of the range to be inserted.

last
: The end of the range to be inserted.

Throws:

Whatever T::T(const T&) throws.
Whatever T::operator = (const T&) throws.

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)

iterator rinsert(iterator pos, const_reference item = value_type());

Insert an element before the specified position.

Precondition:

`pos` is a valid iterator pointing to the `circular_buffer` or its end.

Effect:

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.

Parameter(s):

`pos`
An iterator specifying the position before which the `item` will be inserted.

`item`
The element to 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.

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)

void rinsert(iterator pos, size_type n, const_reference item);

Insert n copies of the item before the specified position.

Precondition:

pos is a valid iterator pointing to the circular_buffer or its end.

Effect:

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.)

Parameter(s):

pos
: An iterator specifying the position where the items will be inserted.

n
: The number of items the to be inserted.

item
: The element whose copies will be inserted.

Throws:

Whatever T::T(const T&) throws.
Whatever T::operator = (const T&) throws.

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)

template <class InputIterator>
void rinsert(iterator pos, InputIterator first, InputIterator last);

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

Precondition:

`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.

Effect:

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.)

Parameter(s):

`pos`
An iterator specifying the position where the range will be inserted.

`first`
The beginning of the range to be inserted.

`last`
The end of the range to be inserted.

Throws:

Whatever `T::T(const T&)` throws.
Whatever `T::operator = (const T&)` throws.

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 `[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)]()

[iterator]() erase([iterator]() pos);

Remove an element at the specified position.

Precondition:

`pos` is a valid iterator pointing to the `circular_buffer` (but not an [end()]()).

Effect:

The element at the position `pos` is removed.

Parameter(s):

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

Returns:

Iterator to the first element remaining beyond the removed element or [end()]() if no such element exists.

Throws:

Whatever `T::operator = (const T&)` throws.

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()

iterator erase(iterator first, iterator last);

Erase the range `[first, last)`.

Precondition:
Valid range `[first, last)`.

Effect:
The elements from the range `[first, last)` are removed. (If `first == last` nothing is removed.)

Parameter(s):

`first`
The beginning of the range to be removed.

`last`
The end of the range to be removed.

Returns:
Iterator to the first element remaining beyond the removed elements or end() if no such element exists.

Throws:
Whatever `T::operator = (const T&)` throws.

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()

iterator rerase(iterator pos);

Remove an element at the specified position.

Precondition:
`pos` is a valid iterator pointing to the `circular_buffer` (but not an end()).

Effect:
The element at the position `pos` is removed.

Parameter(s):

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

Returns:

Iterator to the first element remaining in front of the removed element or begin() if no such element exists.

Throws:
Whatever `T::operator = (const T&)` throws.

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()

---

iterator rerase(iterator first, iterator last);

Erase the range `[first, last)`.

Precondition:
Valid range `[first, last)`.

Effect:
The elements from the range `[first, last)` are removed. (If `first == last` nothing is removed.)

Parameter(s):

`first`
The beginning of the range to be removed.

`last`
The end of the range to be removed.

Returns:
Iterator to the first element remaining in front of the removed elements or begin() if no such element exists.

Throws:
Whatever `T::operator = (const T&)` throws.

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 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()

void erase_begin(size_type n);

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

Precondition:

n <= size()

Effect:

The n elements at the beginning of the circular_buffer will be removed.

Parameter(s):

n

The number of elements to be removed.

Throws:

Whatever T::operator = (const T&) throws. (Does not throw anything in case of scalars.)

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()

void erase_end(size_type n);

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

Precondition:

n <= size()

Effect:

The n elements at the end of the circular_buffer will be removed.

Parameter(s):

n
: The number of elements to be removed.

Throws:
: Whatever `T::operator = (const T&)` throws. (Does not throw anything in case of scalars.)

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()

void clear();

Remove all stored elements from the `circular_buffer`.

Effect:
: size() == 0

Throws:
: Nothing.

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)

Standalone Functions

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

Compare two `circular_buffer`s element-by-element to determine if they are equal.

Parameter(s):

`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.

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

Iterator Invalidation:
Does not invalidate any iterators.

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

Compare two `circular_buffer`s element-by-element to determine if the left one is lesser than the right one.

Parameter(s):

`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.

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

Iterator Invalidation:
Does not invalidate any iterators.

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

Compare two `circular_buffer`s element-by-element to determine if they are non-equal.

Parameter(s):

`lhs`

The `circular_buffer` to compare.

`rhs`
The `circular_buffer` to compare.

Returns:
`!(lhs == rhs)`

Throws:
Nothing.

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>&)]

---

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

Compare two `circular_buffer`s element-by-element to determine if the left one is greater than the right one.

Parameter(s):

`lhs`
The `circular_buffer` to compare.

`rhs`
The `circular_buffer` to compare.

Returns:
`rhs < lhs`

Throws:
Nothing.

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>&)]

---

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

Compare two `circular_buffer`s element-by-element to determine if the left one is lesser or equal to the right one.

Parameter(s):

`lhs`
The `circular_buffer` to compare.

rhs
The `circular_buffer` to compare.

Returns:
`!(rhs < lhs)`

Throws:
Nothing.

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>&)]()

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

Compare two `circular_buffer`s element-by-element to determine if the left one is greater or equal to the right one.

Parameter(s):

lhs
The `circular_buffer` to compare.

rhs
The `circular_buffer` to compare.

Returns:
`!(lhs < rhs)`

Throws:
Nothing.

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>&)]()

**template <class T, class Alloc>
void swap(circular_buffer<T,Alloc>& lhs, circular_buffer<T,Alloc>& rhs);**

Swap the contents of two `circular_buffer`s.

Effect:
`lhs` contains elements of `rhs` and vice versa.

Parameter(s):

lhs
The `circular_buffer` whose content will be swapped with `rhs`.

rhs
The `circular_buffer` whose content will be swapped with `lhs`.

Throws:
Nothing.

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>&)]

Notes

1. A good implementation of smart pointers is included in [Boost].
2. 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.)

See also

[boost::circular_buffer_space_optimized], [std::vector], [std::list], [std::deque]

Acknowledgements

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

Release Notes

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.
