Queue

1.0.0

Generated by Doxygen 1.8.17

Chapter 1

Queue

A modern C++ header-only library of various types of queue

1.1 Table of Contents

1.2 Build Instructions

Each class in the library itself is a single header only. You can use the included CMake in a subdirectory and add the interface library, or just copy the `*.h` files (and license) into your project.

To build and run the unit tests:

```
mkdir build
cd build
cmake -DBUILD_TESTS=ON ..
cmake --build . --config Release
ctest
```

To build the documentation, replace the above `cmake` command with the following:

```
cmake -DBUILD_TESTS=ON -DBUILD_DOCUMENTATION=ON ..
```

1.3 Usage

1.3.1 Concurrent Queue

The `concurrent_queue` class is a templated FIFO (first-in, first-out) queue that is suitable for concurrent use, including cases where there are multiple producers and multiple consumers reading and writing the queue simultaneously. It has an atomic `push`/`pop` interface for enqueueing and dequeueing elements.

1.3.1.1 Example

Run it live on Compiler Explorer.

```
#include <chrono>
#include <future>
#include <iostream>
#include <numeric>
#include <thread>
#include <vector>
#include <concurrent_queue.h>
using namespace std::chrono_literals;
int main()
{
        // Push 100 ints from one deque to another via a set of producer/consumer threads
        // running at different rates via a concurrent queue

        constexpr int    test_size = 100;
        std::deque<int> numbers_in(test_size);
        std::deque<int> numbers_out;
        std::iota(numbers_in.begin(), numbers_in.end(), 0);

        std::cout « "In size: " « numbers_in.size() « std::endl;
        std::cout « "Out size: " « numbers_out.size() « std::endl;

        concurrent_queue<int> queue;

        // produce integers at 200Hz and consume them at 1000 Hz
        auto producer = std::async(std::launch::async, [&queue, &numbers_in]
               {
                   while(!numbers_in.empty())
                   {
                       queue.push(numbers_in.front());
                       numbers_in.pop_front();
                       std::this_thread::sleep_for(5ms);
                   }
                   return std::this_thread::get_id();
               });
        auto consumer = std::async(std::launch::async, [&queue, &numbers_out, test_size]
               {
                   int val;
                   while (numbers_out.size() < test_size)
                   {
                       if (queue.try_pop_for(val, 1ms))
                           numbers_out.push_back(val);
                   }
                   return std::this_thread::get_id();
               });
        auto producer_id = producer.get();
        auto consumer_id = consumer.get();

        std::cout « "In size: " « numbers_in.size() « std::endl;
        std::cout « "Out size: " « numbers_out.size() « std::endl;
}
```

1.3.2 Circular Queue

The `circular_queue` class is a fixed capacity, STL-style, templated circular buffer.

See http://en.wikipedia.org/wiki/Circular_buffer for details of how circular buffers work.

This implementation shares many features of a double-ended queue. This structure allows for the individual elements to be accessed directly through random access iterators, with storage handled automatically by over-writing elements from the opposite end of the container as it grows. As long as elements are added consistently to EITHER the front or the back, once full, the container will expand by over-writing the oldest element. If elements are added to both sides of the buffer, the behavior is effectively user defined.

Therefore, the circular_queue provides a functionality similar to vectors, but with efficient insertion and deletion of elements also at the beginning of the sequence, and not only at its end. Also, like vectors (and unlike dequeues), all elements are stored in contiguous memory locations, and offset access IS allowed.

Both vectors and the circular_queue provide a very similar interface and can be used for similar purposes, but internally both work in quite different ways: While vectors use a single array that needs to be occasionally reallocated for growth, the elements of a circular_queue are over-written once the buffer reaches its maximum capacity, ensuring elements are inserted and accessed in constant time and with a uniform sequential interface (through iterators), as opposed to the vector's amortized constant time. This allows them to grow more efficiently under certain circumstances, especially with very long sequences, where reallocation's become more expensive, and real-time applications where stale data is not as useful as current data.

For operations that involve frequent insertion or removals of elements at positions other than the beginning or the end, circular_queue performs worse and has less consistent iterators and references than lists and forward lists.

1.3.2.1 Example

Run it live on Compiler Explorer.

```
#include <iostream>
#include <circular_queue.h>
int main()
{
    circular_queue<int> q(10);
    for(int i = 0; i < 13; ++i)
    {
        q.push_back(i);
        if(q.full()) std::cout « "Buffer Overflow!" « std::endl;
    }
    for(unsigned int i = 0; i < q.size(); ++i)
    {
        std::cout « "q[" « i « "]: " « q[i] « std::endl;
    }
}
```

Chapter 2

Class Index

2.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 3

File Index

3.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 4

Class Documentation

4.1 circular_queue< T, Alloc > Class Template Reference

Fixed capacity, STL-style, templated circular buffer.

`#include <circular_queue.h>`

Classes

- class const_iterator

  constant iterator for the circular_queue class.
- struct const_queue_pointer

  constant pointer with parity
- class iterator

  iterator for the circular_queue class.
- struct queue_pointer

  pointer with parity

Public Types

- using allocator_type = Alloc

  Type of object used to dynamically allocate memory.
- using difference_type = ptrdiff_t

  A signed integral type, identical to:iterator_traits<iterator>::difference_type.
- using size_type = size_t

  An unsigned integral type that can represent any non-negative value of difference_type.
- using value_type = T

  The template parameter (T), representing the values stored in the container.
- using pointer = T *

  For the default allocator: value_type.*
- using const_pointer = const T *

Type for pointer to constant T object.

- using reference = T &

  Type for data references.

- using const_reference = const T &

  Type for reference to constant T object.

- using reverse_iterator = std::reverse_iterator< iterator >

  Type for reverse iterators.

- using const_reverse_iterator = std::reverse_iterator< const_iterator >

  Type for constant reverse iterators.

Public Member Functions

Constructors

Methods to construct, destruct, and assign the container.

- circular_queue (size_type capacity, const allocator_type &alloc=allocator_type())

  Constructor.

- circular_queue (size_type capacity, const value_type &val, const allocator_type &alloc=allocator_type())

  fill constructor

- template<class InputIterator >
  circular_queue (InputIterator first, InputIterator last, const allocator_type &alloc=allocator_type(), typename std::enable_if<!std::is_integral< InputIterator >::value >::type *=0)

  Range Constructor.

- circular_queue (std::initializer_list< value_type > il, const allocator_type &alloc=allocator_type())

  construct from initializer list

- circular_queue (const circular_queue &other)

  Copy Constructor.

- circular_queue (circular_queue &&other) noexcept

  move constructor

Destructor

- virtual ~circular_queue ()

  destructor

Operators

- circular_queue & operator= (const circular_queue &other)

  Assign content.

- circular_queue & operator= (circular_queue &&other) noexcept

  Assign content.

- circular_queue & operator= (std::initializer_list< value_type > il)

  Assign content.

- reference operator[ ] (size_type n)

  Access element.

- const_reference operator[ ] (size_type n) const

  This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Iterators

Methods to construct iterators to the container.

- iterator begin () noexcept
  Returns iterator to beginning.
- const_iterator begin () const noexcept
  This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.
- iterator end () noexcept
  Returns iterator to end.
- const_iterator end () const noexcept
  This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.
- reverse_iterator rbegin () noexcept
  Returns reverse iterator to beginning.
- const_reverse_iterator rbegin () const noexcept
  This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.
- reverse_iterator rend () noexcept
  Returns reverse iterator to end.
- const_reverse_iterator rend () const noexcept
  This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.
- const_iterator cbegin () const noexcept
  Returns constant iterator to beginning.
- const_iterator cend () const noexcept
  Returns constant iterator to end.
- const_reverse_iterator crbegin () const noexcept
  Returns constant iterator to beginning.
- const_reverse_iterator crend () const noexcept
  Returns constant iterator to end.

Capacity

Methods to determine the capacity of the container

- size_type size () const noexcept
  Return size.
- size_type capacity () const noexcept
  Get buffer capacity.
- size_type max_size () const noexcept
  Return maximum capacity.
- bool empty () const noexcept
  Test whether container is empty.
- bool full () const noexcept
  Test whether container is full.
- void reserve (size_type n)
  Request a change in capacity.
- void resize (size_type n, const value_type &val=value_type())
  Change size.
- void shrink_to_fit ()
  Shrink container to fit.

Element Access

Members that provide access to individual elements contained withing the circular_queue

- reference at (size_type n)
  Access element.
- const_reference at (size_type n) const
  This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.
- reference front () noexcept
  Access first element.
- const_reference front () const noexcept
  This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.
- reference back ()
  Access last element.
- const_reference back () const
  This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Modifiers

Member functions that modify the container.

- template<class InputIterator >
  void assign (InputIterator first, InputIterator last)
  Assign container content range.
- void assign (size_type n, const value_type &val)
  Assign container content by fill.
- void assign (std::initializer_list< value_type > il)
  Assign container contents from initializer list.
- void clear () noexcept
  Clear content.
- void push_back (const value_type &val)
  Add element at the end.
- void push_back (value_type &&val)
  This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.
- void push_front (const value_type &val)
  Add element at beginning.
- void push_front (value_type &&val)
  This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.
- void pop_back () noexcept
  Delete last element.
- void pop_front () noexcept
  Delete first element.
- iterator insert (const_iterator position, const value_type &val)
  Insert (copy) single element.
- iterator insert (const_iterator position, value_type &&val)
  Insert (move) single element.
- iterator insert (const_iterator position, size_type n, const value_type &val)
  Insert (fill) elements.
- template<class InputIterator >
  iterator insert (const_iterator position, InputIterator first, InputIterator last)
  Insert (range) elements.
- iterator insert (const_iterator position, std::initializer_list< value_type > il)
  Insert elements (initializer_list)
- iterator erase (const_iterator position)

Erase element.

- iterator erase (const_iterator first, const_iterator last)

  Erase elements.
- void swap (circular_queue &other) noexcept

  Swap content.
- template<class... Args>
  iterator emplace (const_iterator position, Args &&... args)

  Construct and insert element.
- template<class... Args>
  void emplace_front (Args &&... args)

  Construct and insert element at beginning.
- template<class... Args>
  void emplace_back (Args &&... args)

  Construct and insert element at the end.

Accessors

Access member objects of the container

- allocator_type get_allocator () const noexcept

  Get allocator.
- template<class copyFunctor >
  auto copy (const_iterator first, const_iterator last, value_type *destination, copyFunctor cf) const -> decltype(cf((void *) destination,(void *) first.m_pointer.ptr,((last.m_pointer.ptr - first.m_pointer.ptr) *sizeof(value_type))))

  Copies range to destination buffer using provided copy function.

Protected Member Functions

- queue_pointer increment (queue_pointer p, difference_type n) const noexcept

  increments a pointer
- const_queue_pointer increment (const_queue_pointer p, difference_type n) const noexcept

  This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.
- queue_pointer decrement (queue_pointer p, difference_type n) const noexcept

  decrements a pointer
- const_queue_pointer decrement (const_queue_pointer p, difference_type n, bool &parity) const noexcept

  This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.
- void reallocate (size_type n)

  moves the currently contained data to a new area with size n.
- iterator swapDownTo (const_iterator position, size_type n) noexcept

  Swaps the back of the queue down to position with elements n positions away.
- iterator swapUpToEnd (const_iterator position, size_type n) noexcept

  Moves the given position to the end of the queue by successively swapping it.

Protected Attributes

- Alloc m_alloc

  Allocator class. NOT c-style malloc.
- pointer m_data

  Pointer to the internal data array.
- size_type m_capacity

  capacity of the buffer array.
- queue_pointer m_head

  Points to the logical beginning of the buffer. No memory ownership!
- queue_pointer m_tail

  Points to the logical end of the buffer. No memory ownership!

4.1.1 Detailed Description

template<class T, class Alloc = std::allocator<T>>
class circular_queue< T, Alloc >

Fixed capacity, STL-style, templated circular buffer.

see http://en.wikipedia.org/wiki/Circular_buffer for details of how circular buffers work.

This implementation shares many features of a double-ended queue. This structure allows for the individual elements to be accessed directly through random access iterators, with storage handled automatically by over-writing elements from the opposite end of the container as it grows. As long as elements are added consistently to EITHER the front or the back, once full, the container will expand by over-writing the oldest element. If elements are added to both sides of the buffer, the behavior is effectively user defined.

Therefore, the circular_queue provides a functionality similar to vectors, but with efficient insertion and deletion of elements also at the beginning of the sequence, and not only at its end. Also, like vectors (and unlike dequeues), all elements are stored in contiguous memory locations, and offset access IS allowed.

Both vectors and the circular_queue provide a very similar interface and can be used for similar purposes, but internally both work in quite different ways : While vectors use a single array that needs to be occasionally reallocated for growth, the elements of a circular_queue are over-written once the buffer reaches its maximum capacity, ensuring elements are inserted and accessed in constant time and with a uniform sequential interface (through iterators), as opposed to the vector's amortized constant time. This allows them to grow more efficiently under certain circumstances, especially with very long sequences, where reallocation's become more expensive, and real-time applications where stale data is not as useful as current data.

For operations that involve frequent insertion or removals of elements at positions other than the beginning or the end, circular_queues perform worse and have less consistent iterators and references than lists and forward lists.

This class can be moved very quickly (constant time). It may be slow to copy.
Container properties
Sequence
Elements in sequence containers are ordered in a strict linear sequence. Individual elements are accessed by their position in this sequence.
Fixed-capacity array
Allows direct access to any element in the sequence and provides fast addition / removal of elements at the beginning or the end of the sequence.
Allocator - aware
The container uses an allocator object to dynamically handle its storage needs.

4.1.2 Constructor & Destructor Documentation

4.1.2.1 circular_queue() [1/6]

```
template<class T , class Alloc = std::allocator<T>>
circular_queue< T, Alloc >::circular_queue (
            size_type capacity,
            const allocator_type & alloc = allocator_type() )  [inline], [explicit]
```

Constructor.

Complexity: O(N) with buffer capacity (memory allocation).
Iterator Validity: N/A.
Data Races: N/A.
Exception Safety: Throws std::bad_alloc if allocation fails. In this case, no object is created.

Parameters

in	*capacity*	number of elements that can be stored in the queue.
in	*alloc*	Allocator object. The container keeps and uses an internal copy of this allocator.

4.1.2.2 circular_queue() [2/6]

```
template<class T , class Alloc = std::allocator<T>>
circular_queue< T, Alloc >::circular_queue (
            size_type capacity,
            const value_type & val,
            const allocator_type & alloc = allocator_type() )  [inline], [explicit]
```

fill constructor

Creates an array of size *capacity* and fills it with copies of *val*.
Complexity: O(N) with size.
Iterator Validity: N/A.
Data Races: N/A.
Exception Safety: Throws std::bad_alloc if allocation fails. In this case, no object is created.

Parameters

in	*val*	this value will be copied into all the container elements.
in	*capacity*	size of the container.
in	*alloc*	Allocator object. The container keeps and uses an internal copy of this allocator.

4.1.2.3 circular_queue() [3/6]

```
template<class T , class Alloc = std::allocator<T>>
template<class InputIterator >
circular_queue< T, Alloc >::circular_queue (
            InputIterator first,
            InputIterator last,
            const allocator_type & alloc = allocator_type(),
            typename std::enable_if<!std::is_integral< InputIterator >::value >::type *  = 0 )
[inline], [explicit]
```

Range Constructor.

Creates a container by copying the objects in the range of the InputIterators [first, last). The size and capacity of the container will be equal to the size of the range.
Complexity: O(N) with range.
Iterator Validity: No changes.
Data Races: N/A.
Exception Safety: Throws std::bad_alloc if allocation fails. In this case, no object is created.

Parameters

in	*first*	iterator to the beginning of the range to copy.
in	*last*	iterator to the end (one past the last element) of the range to copy.
in	*alloc*	Allocator object. The container keeps and uses an internal copy of this allocator.

4.1.2.4 circular_queue() [4/6]

```
template<class T , class Alloc = std::allocator<T>>
circular_queue< T, Alloc >::circular_queue (
            std::initializer_list< value_type > il,
            const allocator_type & alloc = allocator_type() )  [inline]
```

construct from initializer list

constructs the circular_queue by copying elements from the initializer_list. The size and capacity of the queue will be equal to the initializer_list size.
Complexity: Linear with size of *il* (constructions).
Iterator Validity: N/A.
Data Races: N/A.
Exception Safety: Throws std::bad_alloc if allocation fails. In this case, no object is created.

Parameters

in	*il*	initializer_list object.
in	*alloc*	Allocator object. The container keeps and uses an internal copy of this

4.1.2.5 circular_queue() [5/6]

```
template<class T , class Alloc = std::allocator<T>>
circular_queue< T, Alloc >::circular_queue (
            const circular_queue< T, Alloc > & other )  [inline]
```

Copy Constructor.

Creates a deep copy of another circular_queue.
Complexity: O(N) with buffer capacity (memory allocation/copy).
Iterator Validity: N/A.
Data Races: N/A.
Exception Safety: Throws std::bad_alloc if allocation fails. In this case, no object is created.

Parameters

in	*other*	buffer to be copied

4.1.2.6 circular_queue() [6/6]

```
template<class T , class Alloc = std::allocator<T>>
circular_queue< T, Alloc >::circular_queue (
            circular_queue< T, Alloc > && other )  [inline], [noexcept]
```

move constructor

constructs a circular_queue and obtains its resources by moving them from another instance.
Complexity: Constant.
Iterator Validity: All iterators to the other container are invalidated.
Data Races: Container *other* is modified.
Exception Safety: No-throw guarantee: this member function never throws exceptions.

Parameters

in	*other*	circular_queue to be moved.

4.1.2.7 ~circular_queue()

```
template<class T , class Alloc = std::allocator<T>>
virtual circular_queue< T, Alloc >::~circular_queue ( )  [inline], [virtual]
```

destructor

Yeah, you know what this is.

4.1.3 Member Function Documentation

4.1.3.1 assign() [1/3]

```
template<class T , class Alloc = std::allocator<T>>
template<class InputIterator >
void circular_queue< T, Alloc >::assign (
            InputIterator first,
            InputIterator last )  [inline]
```

Assign container content range.

Assigns new contents to the container, replacing its current contents, and modifying its size and capacity accordingly.↩ The new contents are elements constructed from each of the elements in the range between first and last, in the same order. The range used is [first, last).
Complexity: Linear in initial and final sizes (destructions, constructions).
Iterator Validity: All iterators are invalidated.
Data Races: The container and all elements are modified. All copied elements are accessed.
Exception Safety: Basic guarantee: if an exception is thrown, the container is in a valid state. If allocator_traits↩ ::construct is not supported with the appropriate arguments for the element constructions, or if the range specified by [first,last) is not valid, it causes undefined behavior. Throws std::length_error if first == last.

Parameters

in	*first*	beginning of the range to copy.
in	*last*	end (exclusive) of the range to be copied.

See also

operator=

4.1.3.2 assign() [2/3]

```
template<class T , class Alloc = std::allocator<T>>
void circular_queue< T, Alloc >::assign (
            size_type n,
            const value_type & val )  [inline]
```

Assign container content by fill.

Assigns new contents to the container, replacing its current contents, and modifying its size and capacity accordingly. The new contents are n elements, each initialized to a copy of val.
Complexity: Linear in initial and final sizes (destructions, constructions).
Iterator Validity: All iterators, pointers, and references are invalidated.
Data Races: The container and all elements are modified. All copied elements are accessed.
Exception Safety: Basic guarantee: if an exception is thrown, the container is in a valid state. If allocator_traits↩ ::construct is not supported with the appropriate arguments for the element constructions, or if the range specified by [first,last) is not valid, it causes undefined behavior. Throws std::length_error if first == last.

Parameters

in	*n*	new container size
in	*val*	value to be copied into each element of the container.

4.1.3.3 assign() [3/3]

```
template<class T , class Alloc = std::allocator<T>>
void circular_queue< T, Alloc >::assign (
            std::initializer_list< value_type > il )  [inline]
```

Assign container contents from initializer list.

Assigns new contents to the container, replacing its current contents, and modifying its size and capacity accordingly. The new contents are copies of the values passed as initializer list, in the same order.
Complexity: Linear in initial and final sizes (destructions, constructions).
Iterator Validity: All iterators, pointers and references related to this container are invalidated.
Data Races: All copied elements are accessed. The container is modified. All contained elements are modified.
Exception Safety: Basic guarantee: if an exception is thrown, the container is in a valid state. If allocator_traits↩ ::construct is not supported with the appropriate arguments for the element constructions it causes undefined behavior. If the initializer list is empty, std::length_error is thrown.

Parameters

in	*il*	An initializer_list object. The compiler will automatically construct such objects from initializer list declarators. Member type value_type is the type of the elements in the container, defined in circular_queue as an alias of its first template parameter (T).

See also

operator=

4.1.3.4 at()

```
template<class T , class Alloc = std::allocator<T>>
reference circular_queue< T, Alloc >::at (
            size_type n )  [inline]
```

Access element.

Returns a reference to the element at position n in the container object. The function automatically checks whether n is within the bounds of valid elements in the container, throwing an out_of_range exception if it is not (i.e., if n is greater or equal than its size). This is in contrast with member operator[], that does not check against bounds.
Complexity: Constant.
Iterator Validity: .
Data Races: .
Exception Safety: .

Parameters

in	*n*	Position of an element in the container. If this is greater than the container size, an exception of type out_of_range is thrown. Notice that the first element has a position of 0 (not 1). Member type size_type is an unsigned integral type.

Returns

The element at the specified position in the container. If the circular_queue object is const-qualified, the function returns a const_reference. Otherwise, it returns a reference.

4.1.3.5 back()

```
template<class T , class Alloc = std::allocator<T>>
reference circular_queue< T, Alloc >::back ( )  [inline]
```

Access last element.

Returns a reference to the last element in the circular_queue. Unlike member end(), which returns an iterator just past this element, this function returns a direct reference. Calling this function on an empty container causes undefined behavior.
Complexity: Constant.
Iterator Validity: No changes.

Data Races: The container is accessed (neither the const nor the non-const versions modify the container). The reference returned can be used to access or modify elements. Concurrently accessing or modifying different elements is safe.
Exception Safety: If the container is not empty, the function never throws exceptions (no-throw guarantee). Otherwise, it causes undefined behavior.

Returns

A reference to the last element in the circular_queue.

4.1.3.6 begin()

```
template<class T , class Alloc = std::allocator<T>>
iterator circular_queue< T, Alloc >::begin ( )  [inline], [noexcept]
```

Returns iterator to beginning.

Returns an iterator to the head (first element) in the container.
Complexity: Constant.
Iterator Validity: No changes.
Data Races: The container is accessed (neither the const nor the non-const versions modify the container). No contained elements are accessed by the call, but the iterator returned can be used to access or modify elements. Concurrently accessing or modifying different elements is safe.
Exception Safety: No-throw guarantee: this member function never throws exceptions. The copy construction or assignment of the returned iterator is also guaranteed to never throw.

Returns

An iterator to the beginning of the sequence container.

4.1.3.7 capacity()

```
template<class T , class Alloc = std::allocator<T>>
size_type circular_queue< T, Alloc >::capacity ( ) const  [inline], [noexcept]
```

Get buffer capacity.

Returns the allocated size of the buffer. For the amount of the buffer in use, see size().
Complexity: Constant.
Iterator Validity: No changes.
Data Races: The container object is accessed.
Exception Safety: No-throw guarantee: This function never throws exceptions.

See also

size()

Returns

capacity of the buffer in terms of how many objects of *value_type* it can contain.

4.1.3.8 cbegin()

```
template<class T , class Alloc = std::allocator<T>>
const_iterator circular_queue< T, Alloc >::cbegin ( ) const  [inline], [noexcept]
```

Returns constant iterator to beginning.

Returns a constant iterator to the head (first element) in the container. A const_iterator is an iterator that points to const content. This iterator can be increased and decreased (unless it is itself also const), just like the iterator returned by begin(), but it cannot be used to modify the contents it points to, even if the container object is not itself const.
Complexity: Constant.
Iterator Validity: No changes.
Data Races: The container is accessed (neither the const nor the non-const versions modify the container). No contained elements are accessed by the call, but the iterator returned can be used to access or modify elements. Concurrently accessing or modifying different elements is safe.
Exception Safety: No-throw guarantee: this member function never throws exceptions. The copy construction or assignment of the returned iterator is also guaranteed to never throw.

Returns

A constant iterator to the beginning of the sequence container.

4.1.3.9 cend()

```
template<class T , class Alloc = std::allocator<T>>
const_iterator circular_queue< T, Alloc >::cend ( ) const  [inline], [noexcept]
```

Returns constant iterator to end.

Returns a constant iterator to the tail (last element) in the container.A const_iterator is an iterator that points to const content. This iterator can be increased and decreased (unless it is itself also const), just like the iterator returned by begin(), but it cannot be used to modify the contents it points to, even if the container object is not itself const.
Complexity: Constant.
Iterator Validity: No changes.
Data Races: The container is accessed (neither the const nor the non-const versions modify the container). No contained elements are accessed by the call, but the iterator returned can be used to access or modify elements. Concurrently accessing or modifying different elements is safe.
Exception Safety: No-throw guarantee: this member function never throws exceptions. The copy construction or assignment of the returned iterator is also guaranteed to never throw.

Returns

A constant iterator to the end of the sequence container.

4.1.3.10 clear()

```
template<class T , class Alloc = std::allocator<T>>
void circular_queue< T, Alloc >::clear ( )  [inline], [noexcept]
```

Clear content.

Removes all elements from the deque (which are destroyed), leaving the container with a size of 0.
Complexity: Linear in size (destructions).
Iterator Validity: All iterators, pointers and references related to this container are invalidated.
Data Races: The container is modified. All contained elements are modified.
Exception Safety: No-throw guarantee: this member function never throws exceptions.

4.1.3.11 copy()

```
template<class T , class Alloc = std::allocator<T>>
template<class copyFunctor >
auto circular_queue< T, Alloc >::copy (
            const_iterator first,
            const_iterator last,
            value_type * destination,
            copyFunctor cf ) const -> decltype(cf((void*) destination, (void*) first.m_pointer.↩
ptr, ((last.m_pointer.ptr - first.m_pointer.ptr) * sizeof(value_type))))   [inline]
```

Copies range to destination buffer using provided copy function.

This function overloads memcpy to handle the internal complexities of copying a circular_queue, which may or may not be wrapped. The resulting array at destination will be unwrapped in either case, meaning logical and physical index[0] will always be the same. Since this function isn't allocator or iterator aware on the destination it's use should be avoided for EVERYTHING except compatibility with C code. In the author's opinion, the ONLY use case for this function is CUDA memory.
THIS MAY NOT WORK if *value_type* does not exhibit bitwise copy semantics.

Parameters

in	*first*	iterator to the first index to be copied.
in	*last*	iterator to the last index to be copied.
out	*destination*	output buffer where the data will be copied to.
in	*copy*	functor used to perform the memory copy. Options may include memcpy, memmove, or bound versions of cudaMemcpy, cudaMemcpyAsync. Must have a signature of copyFunctor(void* destination, void* source, size_t sizeInBytesToCopy)

Returns

forwards the return value of *copyFunctor*.

4.1.3.12 crbegin()

```
template<class T , class Alloc = std::allocator<T>>
const_reverse_iterator circular_queue< T, Alloc >::crbegin ( ) const  [inline], [noexcept]
```

Returns constant iterator to beginning.

Returns a reverse iterator to the reverse beginning (last element) in the container. crbegin points to the element right before the one that would be pointed to by end(). A const_reverse_iterator is an iterator that points to const content. This iterator can be increased and decreased (unless it is itself also const), just like the iterator returned by begin(), but it cannot be used to modify the contents it points to, even if the container object is not itself const.
Complexity: Constant.
Iterator Validity: No changes.
Data Races: The container is accessed (neither the const nor the non-const versions modify the container). No contained elements are accessed by the call, but the iterator returned can be used to access or modify elements. Concurrently accessing or modifying different elements is safe.
Exception Safety: No-throw guarantee: this member function never throws exceptions. The copy construction or assignment of the returned iterator is also guaranteed to never throw.

Returns

A constant iterator to the beginning of the sequence container.

4.1.3.13 crend()

```
template<class T , class Alloc = std::allocator<T>>
const_reverse_iterator circular_queue< T, Alloc >::crend ( ) const  [inline], [noexcept]
```

Returns constant iterator to end.

Returns a reverse iterator pointing to the theoretical element preceding the first element in the container (which is considered its reverse end). A const_reverse_iterator is an iterator that points to const content. This iterator can be increased and decreased (unless it is itself also const), just like the iterator returned by begin(), but it cannot be used to modify the contents it points to, even if the object is not itself const.
Complexity: Constant.
Iterator Validity: No changes.
Data Races: The container is accessed (neither the const nor the non-const versions modify the container). No contained elements are accessed by the call, but the iterator returned can be used to access or modify elements. Concurrently accessing or modifying different elements is safe.
Exception Safety: No-throw guarantee: this member function never throws exceptions. The copy construction or assignment of the returned iterator is also guaranteed to never throw.

Returns

A constant iterator to the end of the sequence container.

4.1.3.14 decrement()

```
template<class T , class Alloc = std::allocator<T>>
queue_pointer circular_queue< T, Alloc >::decrement (
            queue_pointer p,
            difference_type n ) const  [inline], [protected], [noexcept]
```

decrements a pointer

Takes care of wrapping the pointer and keeping parity consistent.

Parameters

in	*p*	pointer to decrement. This function modifies the value of p.
in	*n*	number of elements to increment the pointer
in	*parity*	parity bit associated with the pointer. This function modifies the value of parity.

4.1.3.15 emplace()

```
template<class T , class Alloc = std::allocator<T>>
template<class...  Args>
iterator circular_queue< T, Alloc >::emplace (
            const_iterator position,
            Args &&...  args )  [inline]
```

Construct and insert element.

The container is extended by inserting a new element at position. This new element is constructed in place using args as the arguments for its construction. This effectively increases the container size by one. Double-ended queues are designed to be efficient performing insertions (and removals) from either the end or the beginning of the sequence. Insertions on other positions are usually less efficient than in list or forward_list containers. See emplace_front and emplace_back for member functions that extend the container directly at the beginning or at the end. The element is constructed in-place by calling allocator_traits::construct with args forwarded.
Complexity: Linear in the number of elements between *position* and end().
Iterator Validity: All iterators, references, and pointers from *position* to end() are invalidated.
Data Races: The container is modified. It is NOT safe to concurrently access or modify elements.
Exception Safety: If position is end, there are no changes in the container in case of exception (strong guarantee). Otherwise, the container is guaranteed to end in a valid state (basic guarantee).

Parameters

in	*position*	Position in the container where the new element is inserted. Member type const_iterator is a random access iterator type that points to a constant element.
in	*args*	Arguments forwarded to construct the new element.

See also

emplace_front()

emplace_back()

4.1.3.16 emplace_back()

```
template<class T , class Alloc = std::allocator<T>>
template<class... Args>
void circular_queue< T, Alloc >::emplace_back (
            Args &&... args ) [inline]
```

Construct and insert element at the end.

Inserts a new element at the end of the circular_queue, right after its current last element. This new element is constructed in place using args as the arguments for its construction. This effectively increases the container capacity by one. The element is constructed in-place by calling allocator_traits::construct with args forwarded. A similar member function exists, push_back, which either copies or moves an existing object into the container.
Complexity: Constant, if the contained object can be constructed in constant time.
Iterator Validity: The end iterator is invalidated. If the container was full before the call, the begin iterator is also invalidated. All other iterators, pointers, and references remain valid.
Data Races: The container is modified. If the container was full, the beginning element is destroyed. No other existing elements are accessed.
Exception Safety: If the container is not full, there are no side effects (strong guarantee). Otherwise, if the function throws an exception, no resources are leaked (basic guarantee), however the head of the container will be destroyed even if emplace_back throws and no new object is added to the container.

Parameters

in	*args*	arguments to the constructor of the contained object.

4.1.3.17 emplace_front()

```
template<class T , class Alloc = std::allocator<T>>
template<class... Args>
void circular_queue< T, Alloc >::emplace_front (
            Args &&... args ) [inline]
```

Construct and insert element at beginning.

Inserts a new element at the beginning of the container, right before its current first element. This new element is constructed in place using args as the arguments for its construction.This effectively increases the container size by one. The element is constructed in-place by calling allocator_traits::construct with args forwarded. A similar member function exists, push_front, which either copies or moves an existing object into the container.

Complexity: Constant.
Iterator Validity: The begin() iterator is invalidated. If the container was full before the call, the end iterator is also invalidated. All other iterators, pointers, and references remain valid.
Data Races: The container is modified. If the container was full, the last element is destroyed. No other existing elements are accessed.
Exception Safety: If the container is not full, there are no side effects (strong guarantee). Otherwise, if the function throws an exception, no resources are leaked (basic guarantee), however the head of the container will be destroyed even if emplace_back throws and no new object is added to the container.

Parameters

in	*args*	arguments to the constructor of the contained object.

4.1.3.18 empty()

```
template<class T , class Alloc = std::allocator<T>>
bool circular_queue< T, Alloc >::empty ( ) const  [inline], [noexcept]
```

Test whether container is empty.

Returns whether the container is empty (i.e. whether its size is 0).
Complexity: Constant.
Iterator Validity: No changes.
Data Races: The container object is accessed, but no elements are accessed.
Exception Safety: No-throw guarantee: This function never throws exceptions.

Returns

true if the buffer is empty, false otherwise.

4.1.3.19 end()

```
template<class T , class Alloc = std::allocator<T>>
iterator circular_queue< T, Alloc >::end ( )  [inline], [noexcept]
```

Returns iterator to end.

Returns an iterator to the tail (last element) in the container.
Complexity: Constant.
Iterator Validity: No changes.
Data Races: The container is accessed (neither the const nor the non-const versions modify the container). No contained elements are accessed by the call, but the iterator returned can be used to access or modify elements. Concurrently accessing or modifying different elements is safe.
Exception Safety: No-throw guarantee: this member function never throws exceptions. The copy construction or assignment of the returned iterator is also guaranteed to never throw.

Returns

An iterator to the end of the sequence container.

4.1.3.20 erase() [1/2]

```
template<class T , class Alloc = std::allocator<T>>
iterator circular_queue< T, Alloc >::erase (
            const_iterator first,
            const_iterator last )  [inline]
```

Erase elements.

Removes from the deque container the range of elements ([*first*,*last*)). This effectively reduces the container size by the number of elements removed, which are destroyed.
Double-ended queues are designed to be efficient removing (and inserting) elements at either the end or the beginning of the sequence. Removals on other positions are usually less efficient than in list or forward_list containers.
Complexity: Linear in the distance from *position* to end() times the number of elements erased.
Iterator Validity: All iterators, pointers, and references to elements between *position* and end() are invalidated.
Data Races: If the erasure happens at the end of the sequence, only the erased elements are modified. Otherwise it is not safe to access or modify elements.
Exception Safety: If the removed elements include the the last element in the container, no exceptions are thrown (no-throw guarantee). Otherwise, the container is guaranteed to end in a valid state (basic guarantee): Copying or moving elements while relocating them may throw. Invalid ranges produce undefined behavior.

Parameters

in	*first*	iterator to beginning of range to erase from the container
in	*last*	iterator to one-past-last element of range to erase from the container

Returns

An iterator pointing to the new location of the element that followed the last element erased by the function call. This is the container end if the operation erased the last element in the sequence.

4.1.3.21 erase() [2/2]

```
template<class T , class Alloc = std::allocator<T>>
iterator circular_queue< T, Alloc >::erase (
            const_iterator position )  [inline]
```

Erase element.

Removes from deque container a single element (*position*). This effectively reduces the container size by the number of elements removed, which are destroyed.
Double-ended queues are designed to be efficient removing (and inserting) elements at either the end or the beginning of the sequence. Removals on other positions are usually less efficient than in list or forward_list containers.
Complexity: Linear in the distance from *position* to end().
Iterator Validity: All iterators, pointers, and references to elements between *position* and end() are invalidated.
Data Races: If the erasure happens at the end of the sequence, only the erased elements are modified. Otherwise it is not safe to access or modify elements.
Exception Safety: If the removed elements include the the last element in the container, no exceptions are thrown (no-throw guarantee). Otherwise, the container is guaranteed to end in a valid state (basic guarantee): Copying or moving elements while relocating them may throw. Invalid ranges produce undefined behavior.

Parameters

in	*position*	Iterator pointing to a single element to be removed from the container.

Returns

An iterator pointing to the new location of the element that followed the last element erased by the function call. This is the container end if the operation erased the last element in the sequence.

4.1.3.22 front()

```
template<class T , class Alloc = std::allocator<T>>
reference circular_queue< T, Alloc >::front ( )  [inline], [noexcept]
```

Access first element.

Returns a reference to the first element in the circular_queue. Unlike member begin(), which returns an iterator to this same element, this function returns a direct reference. Calling this function on an empty container causes undefined behavior.
Complexity: Constant.
Iterator Validity: No changes.
Data Races: The container is accessed (neither the const nor the non-const versions modify the container). The reference returned can be used to access or modify elements. Concurrently accessing or modifying different elements is safe.
Exception Safety: If the container is not empty, the function never throws exceptions (no-throw guarantee). Otherwise, it causes undefined behavior.

Returns

A reference to the first element in the circular_queue.

4.1.3.23 full()

```
template<class T , class Alloc = std::allocator<T>>
bool circular_queue< T, Alloc >::full ( ) const  [inline], [noexcept]
```

Test whether container is full.

Returns whether the container is empty (i.e. whether its size is equal to it's capacity).
Complexity: Constant.
Iterator Validity: No changes.
Data Races: The container object is accessed, but no elements are accessed.
Exception Safety: .

Returns

bool

4.1.3.24 get_allocator()

```
template<class T , class Alloc = std::allocator<T>>
allocator_type circular_queue< T, Alloc >::get_allocator ( ) const  [inline], [noexcept]
```

Get allocator.

Returns a copy of the allocator object associated with the container.
Complexity: Constant.
Iterator Validity: No changes.
Data Races: The container is accessed. No contained elements are accessed: concurrently accessing or modifying them is safe.
Exception Safety: No-throw guarantee: this member function never throws exceptions. Copying any instantiation of the default allocator is also guaranteed to never throw.

Returns

the allocator.

4.1.3.25 increment()

```
template<class T , class Alloc = std::allocator<T>>
queue_pointer circular_queue< T, Alloc >::increment (
            queue_pointer p,
            difference_type n ) const  [inline], [protected], [noexcept]
```

increments a pointer

Takes care of wrapping the pointer and keeping parity consistent.

Parameters

in	*p*	pointer to increment. This function modifies the value of p.
in	*n*	number of elements to increment the pointer
in	*parity*	parity bit associated with the pointer. This function modifies the value of parity.

4.1.3.26 insert() [1/5]

```
template<class T , class Alloc = std::allocator<T>>
iterator circular_queue< T, Alloc >::insert (
            const_iterator position,
            const value_type & val )  [inline]
```

Insert (copy) single element.

The container is extended by inserting an element before the element at the specified position. This effectively increases the container size by 1. If the container is full, the element at begin() is overwritten. Makes at most 1 copy of val.
Double-ended queues are designed to be efficient performing insertions (and removals) from either the end or the beginning of the sequence. Insertions on other positions are usually less efficient than in list or forward_list containers.
Complexity: Linear on the number of elements inserted (copy/move construction) plus an additional linear in the number of elements between position and the end of the container.
Iterator Validity: All iterators, pointers and references from position to the end of the queue are invalidated.
Data Races: The container is modified. It is not safe to concurrently access elements.
Exception Safety: The container is guaranteed to end in a valid state (basic guarantee). If allocator_traits::construct is not supported with the appropriate arguments for the element constructions, or if an invalid position or range is specified, it causes undefined behavior.

Parameters

in	*position*	Position in the container where the new elements are inserted. Iterator is a member type, defined as a random access iterator type that points to elements.
in	*val*	Value to be copied (or moved) to the inserted elements. Member type value_type is the type of the elements in the container, defined in deque as an alias of its first template parameter (T).

See also

push_back()

push_front()

Returns

An iterator that points to the newly inserted element.

4.1.3.27 insert() [2/5]

```
template<class T , class Alloc = std::allocator<T>>
template<class InputIterator >
iterator circular_queue< T, Alloc >::insert (
            const_iterator position,
            InputIterator first,
            InputIterator last )  [inline]
```

Insert (range) elements.

The container is extended by inserting new elements before the element at the specified position. This effectively increases the container size by the amount of elements inserted. If this size is greater than the current capacity, the elements at the beginning of the queue will be over-written. This operation will cause at most capacity() copies. The range of copied values is [first, last).
Double-ended queues are designed to be efficient performing insertions (and removals) from either the end or the

beginning of the sequence. Insertions on other positions are usually less efficient than in list or forward_list containers.
Complexity: Linear on the number of elements inserted (copy/move construction) plus an additional linear in the number of elements between position and the end of the container.
Iterator Validity: All iterators, pointers and references are invalidated.
Data Races: The container is modified. It is not safe to concurrently access elements.
Exception Safety: The container is guaranteed to end in a valid state (basic guarantee). If allocator_traits::construct is not supported with the appropriate arguments for the element constructions, or if an invalid position or range is specified, it causes undefined behavior.

Parameters

in	*position*	the fill will be inserted before the element at this position.
in	*first*	iterator to beginning of range to insert. InputIterator can be an iterator to any type of container, as long as it is at least a forward iterator.
in	*last*	iterator to one-past-end of range to insert. InputIterator can be an iterator to any type of container, as long as it is at least a forward iterator. type of the elements in the container, defined in circular_queue as an alias of its first template parameter (T).

See also

push_back()
push_front()

Returns

An iterator that points to the first of the newly inserted elements.

4.1.3.28 insert() [3/5]

```
template<class T , class Alloc = std::allocator<T>>
iterator circular_queue< T, Alloc >::insert (
            const_iterator position,
            size_type n,
            const value_type & val )  [inline]
```

Insert (fill) elements.

The container is extended by inserting new elements before the element at the specified position. This effectively increases the container size by the amount of elements inserted. If this size is greater than the current capacity, the elements at the beginning of the queue will be over-written. This operation will cause at most n copies of *val*.
Double-ended queues are designed to be efficient performing insertions (and removals) from either the end or the beginning of the sequence. Insertions on other positions are usually less efficient than in list or forward_list containers.
Complexity: O(m∗n) where m is the current size and m is the number of elements inserted.
Iterator Validity: All iterators, pointers and references are invalidated.
Data Races: The container is modified. It is not safe to concurrently access elements.
Exception Safety: The container is guaranteed to end in a valid state (basic guarantee). If allocator_traits::construct is not supported with the appropriate arguments for the element constructions, or if an invalid position or range is specified, it causes undefined behavior.

Parameters

in	*position*	the fill will be inserted before the element at this position.
in	*n*	Number of elements to insert. Each element is initialized to a copy of val.
in	*val*	Value to be copied to the inserted elements. Member type value_type is the type of the elements in the container, defined in circular_queue as an alias of its first template parameter (T).

See also

push_back()

push_front()

Returns

An iterator that points to the first of the newly inserted elements.

4.1.3.29 insert() [4/5]

```
template<class T , class Alloc = std::allocator<T>>
iterator circular_queue< T, Alloc >::insert (
            const_iterator position,
            std::initializer_list< value_type > il )  [inline]
```

Insert elements (initializer_list)

The container is extended by inserting new elements before the element at the specified position. This effectively increases the container size by the amount of elements inserted. If this size is greater than the current capacity, the elements at the beginning of the queue will be over-written. This operation will cause at most capacity() copies.
Double-ended queues are designed to be efficient performing insertions (and removals) from either the end or the beginning of the sequence. Insertions on other positions are usually less efficient than in list or forward_list containers.
Complexity: Linear on the number of elements inserted (copy/move construction) plus an additional linear in the number of elements between position and the end of the container.
Iterator Validity: All iterators, pointers and references are invalidated.
Data Races: The container is modified. It is not safe to concurrently access elements.
Exception Safety: The container is guaranteed to end in a valid state (basic guarantee). If allocator_traits::construct is not supported with the appropriate arguments for the element constructions, or if an invalid position or range is specified, it causes undefined behavior.

Parameters

in	*position*	the fill will be inserted before the element at this position.
in	*il*	initializer_list object, filled with values to insert into the container.

See also

push_back()

push_front()

Returns

An iterator that points to the first of the newly inserted elements.

4.1.3.30 insert() [5/5]

```
template<class T , class Alloc = std::allocator<T>>
iterator circular_queue< T, Alloc >::insert (
            const_iterator position,
            value_type && val )  [inline]
```

Insert (move) single element.

The container is extended by inserting an element before the element at the specified position. This effectively increases the container size by 1. If the container is full, the element at begin() is overwritten. Makes no copies of val, and val must be *move-assignable*.
Double-ended queues are designed to be efficient performing insertions (and removals) from either the end or the beginning of the sequence. Insertions on other positions are usually less efficient than in list or forward_list containers.
Complexity: Linear on the number of elements inserted (copy/move construction) plus an additional linear in the number of elements between position and the end of the container.
Iterator Validity: All iterators, pointers and references from position to the end of the queue are invalidated.
Data Races: The container is modified. It is not safe to concurrently access elements.
Exception Safety: The container is guaranteed to end in a valid state (basic guarantee). If allocator_traits::construct is not supported with the appropriate arguments for the element constructions, or if an invalid position or range is specified, it causes undefined behavior.

Parameters

in	*position*	Position in the container where the new elements are inserted. Iterator is a member type, defined as a random access iterator type that points to elements.
in	*val*	Value to be copied (or moved) to the inserted elements. Member type value_type is the type of the elements in the container, defined in deque as an alias of its first template parameter (T).

See also

push_back()

push_front()

Returns

An iterator that points to the newly inserted element.

4.1.3.31 max_size()

```
template<class T , class Alloc = std::allocator<T>>
size_type circular_queue< T, Alloc >::max_size ( ) const  [inline], [noexcept]
```

Return maximum capacity.

Returns the maximum number of elements that the container can hold. This is the maximum potential capacity the container can reach due to known system or library implementation limitations, but the container is by no means guaranteed to be able to reach that capacity: it can still fail to allocate storage at any point before that capacity is reached.
Complexity: Constant.
Iterator Validity: No changes.
Data Races: The container is accessed, but no contained elements are accessed. Concurrently accessing or modifying them is safe.
Exception Safety: No-throw guarantee: This function never throws exceptions.

Returns

Maximum number of elements of type *value_type* that can be stored in the container.

4.1.3.32 operator=() [1/3]

```
template<class T , class Alloc = std::allocator<T>>
circular_queue& circular_queue< T, Alloc >::operator= (
            circular_queue< T, Alloc > && other )  [inline], [noexcept]
```

Assign content.

moves all the elements from *other* into the container (with *other* in an unspecified but valid state).
Complexity: Linear in Size (destructions).
Iterator Validity: All iterators, references and pointers related to this container before the call are invalidated.
Data Races: All copied elements are accessed. The container and all its elements are modified.
Exception Safety: Basic guarantee: if an exception is thrown, the container is in a valid state. If allocator_traits<↩
::construct is not supported with the appropriate arguments for the element constructions, or if value_type is not move assignable, it causes undefined behavior.

Parameters

in	*other*	container to move

See also

assign()

Returns

*this

4.1.3.33 operator=() [2/3]

```
template<class T , class Alloc = std::allocator<T>>
circular_queue& circular_queue< T, Alloc >::operator= (
            const circular_queue< T, Alloc > & other )  [inline]
```

Assign content.

copies all the elements from *other* into the container (with *other* preserving its contents).
Complexity: Linear in initial and final sizes (destructions, copy constructions).
Iterator Validity: All iterators, references and pointers related to this container before the call are invalidated.
Data Races: All copied elements are accessed. The container and all its elements are modified.
Exception Safety: Basic guarantee: if an exception is thrown, the container is in a valid state. If allocator_traits↩ ::construct is not supported with the appropriate arguments for the element constructions, or if value_type is not copy assignable, it causes undefined behavior.

Parameters

in	*other*	container to copy

See also

Returns

*this

4.1.3.34 operator=() [3/3]

```
template<class T , class Alloc = std::allocator<T>>
circular_queue& circular_queue< T, Alloc >::operator= (
            std::initializer_list< value_type > il )  [inline]
```

Assign content.

Assigns new contents to the container, replacing its current contents, and modifying its size accordingly. The new contents are copies of the values passed as initializer list, in the same order.
Complexity: Linear in initial and final sizes (destructions, constructions).

Iterator Validity: All iterators, references and pointers related to this container before the call are invalidated. **Data Races:** All copied elements are accessed. The container and all its elements are modified.
Exception Safety: Basic guarantee: if an exception is thrown, the container is in a valid state. If allocator_traits↩ ::construct is not supported with the appropriate arguments for the element constructions, or if value_type is not move assignable, it causes undefined behavior.

Parameters

in	*il*	initializer list object.

Returns

circular_queue&

4.1.3.35 operator[]()

```
template<class T , class Alloc = std::allocator<T>>
reference circular_queue< T, Alloc >::operator[] (
            size_type n )  [inline]
```

Access element.

Returns a reference to the element at position n in the container. A similar member function, at(), has the same behavior as this operator function, except that at() is bound-checked and signals if the requested position is out of range by throwing an out_of_range exception. Portable programs should never call this function with an argument n that is out of range, since this causes undefined behavior.
Complexity: Constant.
Iterator Validity: No changes.
Data Races: The container is accessed (neither the const nor the non-const versions modify the container). The reference returned can be used to access or modify elements. Concurrently accessing or modifying different elements is safe.
Exception Safety: If the container size is greater than n, the function never throws exceptions (no-throw guarantee). Otherwise, the behavior is undefined.

Parameters

in	*n*	Position of an element in the container. Notice that the first element has a position of 0 (not 1). Member type size_type is an unsigned integral type.

Returns

The element at the specified position in the circular_queue.

4.1.3.36 pop_back()

```
template<class T , class Alloc = std::allocator<T>>
void circular_queue< T, Alloc >::pop_back ( )  [inline], [noexcept]
```

Delete last element.

Removes the last element in the circular_queue container, effectively reducing its size by one. This destroys the removed element. Popping an empty queue results in undefined behavior.
Complexity: Constant.
Iterator Validity: The iterators, pointers and references referring to the removed element are invalidated. Iterators, pointers and references referring to other elements that have not been removed are guaranteed to keep referring to the same elements they were referring to before the call.
Data Races: The container is modified. The last element is modified. Concurrently accessing or modifying other elements is safe (although see iterator validity above).
Exception Safety: If the container is not empty, the function never throws exceptions (no-throw guarantee). Otherwise, it causes undefined behavior.

4.1.3.37 pop_front()

```
template<class T , class Alloc = std::allocator<T>>
void circular_queue< T, Alloc >::pop_front ( )  [inline], [noexcept]
```

Delete first element.

Removes the first element in the circular_queue container, effectively reducing its size by one. This destroys the removed element. Popping an empty queue results in undefined behavior.
Complexity: Constant.
Iterator Validity: The iterators, pointers and references referring to the removed element are invalidated. Iterators, pointers and references referring to other elements that have not been removed are guaranteed to keep referring to the same elements they were referring to before the call.
Data Races: The container is modified. The first element is modified. Concurrently accessing or modifying other elements is safe (although see iterator validity above).
Exception Safety: If the container is not empty, the function never throws exceptions (no-throw guarantee). Otherwise, it causes undefined behavior.

4.1.3.38 push_back()

```
template<class T , class Alloc = std::allocator<T>>
void circular_queue< T, Alloc >::push_back (
            const value_type & val )  [inline]
```

Add element at the end.

Adds a new element at the end of the container, after its current last element. The content of val is copied (or moved) to the new element. This effectively increases the container size by one. If the circular_queue is full, it will cause the element at the front of the queue to be overwritten.
Complexity: Constant.
Iterator Validity: Only the end iterator is invalidated, and all iterators, pointers and references to elements are guaranteed to keep referring to the same elements they were referring to before the call.
Data Races: The container is modified. If full, the first element in the queue is modified. Concurrently accessing or modifying other elements is safe.
Exception Safety: The container is guaranteed to end in a valid state (basic guarantee). If the element is copyable or no-throw movable, then there are no side effects (strong guarantee). If the container is not full, then there are no side effects (strong guarantee). If the container is full, and an exception occurs, the first element in the container will be destroyed. If allocator_traits::construct is not supported with val as argument, it causes undefined behavior.

Parameters

in	*val*	value to copy/move into the end of the container. Member type value_type is the type of the elements in the container, defined in vector as an alias of its first template parameter (T).

4.1.3.39 push_front()

```
template<class T , class Alloc = std::allocator<T>>
void circular_queue< T, Alloc >::push_front (
            const value_type & val )  [inline]
```

Add element at beginning.

Adds a new element at the beginning of the container, right before its current first element. The content of val is copied (or moved) to the new element. This effectively increases the container size by one. If the circular_queue is full, it will cause the element at the end of the queue to be overwritten.
Complexity: Constant.
Iterator Validity: Only the begin iterator is invalidated, and all iterators, pointers and references to elements are guaranteed to keep referring to the same elements they were referring to before the call.
Data Races: The container is modified. If full, the last element in the queue is modified. Concurrently accessing or modifying other elements is safe.
Exception Safety: The container is guaranteed to end in a valid state (basic guarantee). If the element is copyable or no-throw movable, then there are no side effects (strong guarantee). If allocator_traits::construct is not supported with val as argument, it causes undefined behavior.

Parameters

in	*val*	value to copy/move into the end of the container. Member type value_type is the type of the elements in the container, defined in vector as an alias of its first template parameter (T).

4.1.3.40 rbegin()

```
template<class T , class Alloc = std::allocator<T>>
reverse_iterator circular_queue< T, Alloc >::rbegin ( )  [inline], [noexcept]
```

Returns reverse iterator to beginning.

@function rbegin

Returns a reverse iterator to the reverse beginning (last element) in the container. rbegin points to the element right before the one that would be pointed to by end().
Complexity: Constant.
Iterator Validity: No changes.

Data Races: The container is accessed (neither the const nor the non-const versions modify the container). No contained elements are accessed by the call, but the iterator returned can be used to access or modify elements. Concurrently accessing or modifying different elements is safe.
Exception Safety: No-throw guarantee: this member function never throws exceptions. The copy construction or assignment of the returned iterator is also guaranteed to never throw.

Returns

A reverse iterator to the beginning of the sequence container.

4.1.3.41 reallocate()

```
template<class T , class Alloc = std::allocator<T>>
void circular_queue< T, Alloc >::reallocate (
            size_type n )  [inline], [protected]
```

moves the currently contained data to a new area with size n.

O(n) complexity. Unwraps the buffer as a side effect.

Parameters

in	*n*	size of new allocation

4.1.3.42 rend()

```
template<class T , class Alloc = std::allocator<T>>
reverse_iterator circular_queue< T, Alloc >::rend ( )  [inline], [noexcept]
```

Returns reverse iterator to end.

Returns a reverse iterator pointing to the theoretical element preceding the first element in the container (which is considered its reverse end).
Complexity: Constant.
Iterator Validity: No changes.
Data Races: The container is accessed (neither the const nor the non-const versions modify the container). No contained elements are accessed by the call, but the iterator returned can be used to access or modify elements. Concurrently accessing or modifying different elements is safe.
Exception Safety: No-throw guarantee: this member function never throws exceptions. The copy construction or assignment of the returned iterator is also guaranteed to never throw.

Returns

A reverse iterator to the end of the sequence container.

4.1.3.43 reserve()

```
template<class T , class Alloc = std::allocator<T>>
void circular_queue< T, Alloc >::reserve (
            size_type n )  [inline]
```

Request a change in capacity.

Requests that the vector capacity be at least enough to contain n elements. If n is greater than the current capacity, the function causes the container to reallocate its storage increasing its capacity to n. In all other cases, the function call does not cause a reallocation and the vector capacity is not affected.
Complexity: If a reallocation happens, linear in size at most..
Iterator Validity: If a reallocation happens, all iterators, pointers and references related to the container are invalidated. Otherwise, they all keep referring to the same elements they were referring to before the call.
Data Races: If a reallocation happens, the container and all its contained elements are modified.
Exception Safety: If no reallocations happen or if the type of the elements has either a non-throwing move constructor or a copy constructor, there are no changes in the container in case of exception (strong guarantee). Otherwise, the container is guaranteed to end in a valid state (basic guarantee). Under the basic guarantee, if this function throws, it's possible that all the container contents will be destroyed as a side-effect. The function throws bad_alloc if it fails to allocate the new size.

Parameters

in	*n*	Minimum capacity for the container.

4.1.3.44 resize()

```
template<class T , class Alloc = std::allocator<T>>
void circular_queue< T, Alloc >::resize (
            size_type n,
            const value_type & val = value_type() )  [inline]
```

Change size.

Changes *both* the size and capacity of the container, resizing the container so that it contains n elements. If n is smaller than the current container size, the content is reduced to its first n elements, removing those beyond (and destroying them). If n is greater than the current container size, the content is expanded by inserting at the end as many elements as needed to reach a size of n. If val is specified, the new elements are initialized as copies of val, otherwise, they are value-initialized. Regardless of the value of n (except n == size()), an automatic reallocation of the allocated storage space takes place. Notice that this function changes the actual content of the container by inserting or erasing elements from it.
Complexity: Linear on the number of elements inserted/erased (constructions/destructions). If a reallocation happens, the reallocation is itself up to linear in the entire container size.
Iterator Validity: Resizing causes reallocation, and so all iterators, pointers and references related to this container are also invalidated.
Data Races: The container is modified. If a reallocation happens, all contained elements are modified.
Exception Safety: If n is less than or equal to the size of the container, the function never throws exceptions (no-throw

guarantee). If n is greater and a reallocation happens, there are no changes in the container in case of exception (strong guarantee) if the type of the elements is either copyable or no-throw movable. Otherwise, if an exception is thrown, the container is left with a valid state (basic guarantee).

Parameters

in	*n*	New container size, expressed in number of elements. Member type *size_type* is an unsigned integral type.
in	*val*	Object whose content is copied to the added elements in case that n is greater than the current container size. If not specified, the default constructor is used instead.

4.1.3.45 shrink_to_fit()

```
template<class T , class Alloc = std::allocator<T>>
void circular_queue< T, Alloc >::shrink_to_fit ( )  [inline]
```

Shrink container to fit.

Requests the container to reduce its capacity to fit its size. Unlike vector, the request is binding. This will never cause a reallocation, and has no effect on the size or element contents.
Complexity: At most, linear in container size..
Iterator Validity: All iterators, pointers and references related to the container are invalidated.
Data Races: The container is modified. All contained objects are moved.
Exception Safety: If the type of the elements is either copyable or no-throw movable, there are no changes in the container in case of exception (strong guarantee). Otherwise, if an exception is thrown, the container is left with a valid state (basic guarantee).

4.1.3.46 size()

```
template<class T , class Alloc = std::allocator<T>>
size_type circular_queue< T, Alloc >::size ( ) const  [inline], [noexcept]
```

Return size.

Returns the number of elements in the container.
Complexity: Constant.
Iterator Validity: No changes.
Data Races: The container is accessed. No contained elements are accessed: concurrently accessing or modifying them is safe.
Exception Safety: No-throw guarantee: this member function never throws exceptions..

Returns

The number of elements in the container. Member type size_type is an unsigned integral type.

4.1.3.47 swap()

```
template<class T , class Alloc = std::allocator<T>>
void circular_queue< T, Alloc >::swap (
            circular_queue< T, Alloc > & other )  [inline], [noexcept]
```

Swap content.

Exchanges the content of the container by the content of *other*, which is another circular_queue object containing elements of the same type. Sizes may differ. After the call to this member function, the elements in this container are those which were in *other* before the call, and the elements of *other* are those which were in *this*. All iterators, references and pointers remain valid for the swapped objects. Notice that a non-member function exists with the same name, swap, overloading that algorithm with an optimization that behaves like this member function.
Complexity: Constant.
Iterator Validity: All iterators, pointers and references referring to elements in both containers remain valid, and are now referring to the same elements they referred to before the call, but in the other container, where they now iterate.
Data Races: Both the container and *other* are modified. No contained elements are accessed by the call.
Exception Safety: If the allocators in both containers compare equal, or if their allocator traits indicate that the allocators shall propagate, the function never throws exceptions (no-throw guarantee). Otherwise, it causes undefined behavior.

Parameters

in	*other*	container to swap with *this*.

4.1.3.48 swapDownTo()

```
template<class T , class Alloc = std::allocator<T>>
iterator circular_queue< T, Alloc >::swapDownTo (
            const_iterator position,
            size_type n )  [inline], [protected], [noexcept]
```

Swaps the back of the queue down to position with elements n positions away.

No-throw. O(n) complexity.

Parameters

in	*position*	position to swap down to from the end
in	*n*	number of elements

Returns

iterator to position of last element swapped.

4.1.3.49 swapUpToEnd()

```
template<class T , class Alloc = std::allocator<T>>
iterator circular_queue< T, Alloc >::swapUpToEnd (
            const_iterator position,
            size_type n )  [inline], [protected], [noexcept]
```

Moves the given position to the end of the queue by successively swapping it.

No-throw. O(n) complexity.

Parameters

in	*position*	position to swap up to the end from
in	*n*	number of elements

Returns

iterator to position of first element swapped.

The documentation for this class was generated from the following file:

- include/circular_queue.h

4.2 concurrent_queue< T, Queue, Alloc > Class Template Reference

The `concurrent_queue` class is a sequence container class that allows first-in, first-out access to its elements.

`#include <concurrent_queue.h>`

Public Types

- typedef T value_type

  A type that represents the data type stored in a concurrent queue.
- typedef Alloc allocator_type

  A type that represents the allocator class for the concurrent queue.
- typedef value_type & reference

  A type that provides a reference to an element stored in a concurrent queue.
- typedef std::condition_variable_any condition_type

  A type that provides a waitable condition of the concurrent queue.
- typedef std::allocator_traits< allocator_type >::pointer pointer

  A type that provides a pointer to an element stored in a concurrent queue.
- typedef std::allocator_traits< allocator_type >::const_pointer const_pointer

  A type that provides a const pointer to an element stored in a concurrent queue.
- typedef Queue< T, Alloc > queue_type

A type that represents the underlying non-thread-safe data structure for the concurrent queue.

- typedef queue_type::iterator iterator

  A type that represents a non-thread-safe iterator over the elements in a concurrent queue.
- typedef queue_type::const_iterator const_iterator

  A type that represents a non-thread-safe const iterator over elements in a concurrent queue.
- typedef std::reverse_iterator< iterator > reverse_iterator

  A type that represents a reverse non-thread-safe iterator over the elements in a concurrent queue.
- typedef std::reverse_iterator< const_iterator > const_reverse_iterator

  A type that represents a reverse non-thread-safe const iterator over elements in a concurrent queue.
- typedef std::shared_timed_mutex mutex_type

  A type that represents the mutex protecting the concurrent queue.
- typedef std::shared_lock< mutex_type > read_lock_type

  A type representing a lock on the concurrent queue's mutex which is sufficient to read data in a thread-safe manner.
- typedef std::unique_lock< mutex_type > write_lock_type

  A type representing a lock on the concurrent queue's mutex which is sufficient to write data in a thread-safe manner.
- typedef std::iterator_traits< iterator >::difference_type difference_type

  A type that provides the signed distance between two elements in a concurrent queue.
- typedef size_t size_type

  A type that counts the number of elements in a concurrent queue.

Public Member Functions

Constructors

- concurrent_queue ()=default

  Default Constructor.
- concurrent_queue (const allocator_type &alloc)

  Default Constructor.
- concurrent_queue (size_type n, const allocator_type &alloc=allocator_type())

  Fill Constructor.
- concurrent_queue (size_type n, const value_type &val, const allocator_type &alloc=allocator_type())

  Fill Constructor.
- template<typename InputIterator >
  concurrent_queue (InputIterator first, InputIterator last, const allocator_type &alloc=allocator_type())

  Range Constructor.
- concurrent_queue (const concurrent_queue &other)

  Copy Constructor.
- concurrent_queue (const concurrent_queue &other, const allocator_type &alloc)

  Copy Constructor with allocator.
- concurrent_queue (concurrent_queue &&other) noexcept

  Move Constructor.
- concurrent_queue (concurrent_queue &&other, const allocator_type &alloc) noexcept

  Move Constructor.
- concurrent_queue (std::initializer_list< T > init, const allocator_type &alloc=allocator_type())

  Initializer List Constructor.

Destructor

- ~concurrent_queue ()=default

  Destructor.

Assignment Operators

- concurrent_queue & operator= (const concurrent_queue &other)
  Copy Assignment Operator.
- concurrent_queue & operator= (concurrent_queue &&other) noexcept
  Move Assignment Operator.

Thread-safe Public Members

These methods are safe for use in situations where the queue will be concurrently accessed by multiple threads.

- void clear () noexcept
  Clears the concurrent queue, destroying any currently enqueued elements.
- template<class... Args>
  void emplace (Args &&... args)
  Constructs a new element in place at the end of the concurrent queue.
- bool empty () const noexcept
  Tests if the concurrent queue is empty at the moment this method is called.
- allocator_type get_allocator () const noexcept
  Returns a copy of the allocator used to construct the concurrent queue.
- void push (const T &value)
  Enqueues an item at tail end of the concurrent queue.
- void push (T &&value)
  Enqueues an item at tail end of the concurrent queue.
- size_t size () const
  Returns the number of items in the queue.
- bool try_pop (T &destination)
  Dequeues an item from the queue if one is available.
- template<class Rep , class Period >
  bool try_pop_for (T &destination, const std::chrono::duration< Rep, Period > &timeout_duration)
  Dequeues an item from the queue if one is available within the specified timeout.

Thread-unsafe Public Members

These methods are not recommended for use in production code and will generate warnings when used. That said, they're helpful for testing and debug, which is why they are included.

- iterator begin ()
  Returns an iterator of type iterator to the beginning of the concurrent queue.
- const_iterator begin () const
  Returns an iterator of type const_iterator to the beginning of the concurrent queue.
- const_iterator cbegin () const
  Returns an iterator of type const_iterator to the beginning of the concurrent queue.
- iterator end ()
  Returns an iterator of type iterator to the end of the concurrent queue.
- const_iterator end () const
  Returns an iterator of type const_iterator to the end of the concurrent queue.
- const_iterator cend () const
  Returns an iterator of type const_iterator to the end of the concurrent queue.

Access Control

These methods provide the ability to lock the concurrent_queue for thread-safe iteration. They are primarily intended for test and debug use, and care must be taken when explicitely locking the queue to avoid deadlock. These methods are NOT recommended for production code, and will produce warnings when used.

- read_lock_type acquire_read_lock () const
  Lock the queue in a manner in which it is safe for multiple threads to read-iterate over it.
- write_lock_type acquire_write_lock ()
  Lock the queue in a manner in which it is safe for multiple threads to write-iterate over it.

Public Attributes

- const typedef value_type & const_reference

  A type that provides a reference to a const element stored in a concurrent queue for reading and performing const operations.

Friends

Equality Operators

- template<class Ty , template< class, class > class Q, class A >
  bool operator== (const concurrent_queue< Ty, Q, A > &lhs, const concurrent_queue< Ty, Q, A > &rhs)

  Equality Operator.
- template<class Ty , template< class, class > class Q, class A >
  bool operator!= (const concurrent_queue< Ty, Q, A > &lhs, const concurrent_queue< Ty, Q, A > &rhs)

  Inequality Operator.

Swap Operator

- template<class Ty , template< class, class > class Q, class A >
  void std::swap (concurrent_queue< Ty, Q, A > &lhs, concurrent_queue< Ty, Q, A > &rhs)

  Swap Operator.

4.2.1 Detailed Description

template<class T, template< class, class > class Queue = std::deque, class Alloc = std::allocator<T>>
class concurrent_queue< T, Queue, Alloc >

The `concurrent_queue` class is a sequence container class that allows first-in, first-out access to its elements.

It enables a limited set of concurrency-safe operations, such as push and try_pop. Here, concurrency-safe means pointers or iterators are always valid. It's not a guarantee of element initialization, or of a particular traversal order.

Template Parameters

T	The data type of the elements to be stored in the queue.
Queue	Underlying concurrent queue data structure. Defaults to `std::deque`.
AllocThe	type that represents the stored allocator object that encapsulates details about the allocation and deallocation of memory for this concurrent queue. This argument is optional and the default value is allocator<T>.

4.2.2 Constructor & Destructor Documentation

4.2.2.1 concurrent_queue() [1/10]

```
template<class T , template< class, class > class Queue = std::deque, class Alloc = std::allocator<↩
T>>
concurrent_queue< T, Queue, Alloc >::concurrent_queue ( )  [inline], [default]
```

Default Constructor.

Constructs an empty container, with no elements.

Parameters

in	*alloc*	optional memory allocator.

4.2.2.2 concurrent_queue() [2/10]

```
template<class T , template< class, class > class Queue = std::deque, class Alloc = std::allocator<↩
T>>
concurrent_queue< T, Queue, Alloc >::concurrent_queue (
            const allocator_type & alloc )  [inline], [explicit]
```

Default Constructor.

Constructs an empty container, with no elements.

Parameters

in	*alloc*	optional memory allocator.

4.2.2.3 concurrent_queue() [3/10]

```
template<class T , template< class, class > class Queue = std::deque, class Alloc = std::allocator<↩
T>>
concurrent_queue< T, Queue, Alloc >::concurrent_queue (
            size_type n,
            const allocator_type & alloc = allocator_type() )  [inline], [explicit]
```

Fill Constructor.

Constructs a container with n elements. Each element is a copy of val (if provided).

Parameters

in	*n*	number of elements
in	*alloc*	optional memory allocator.

4.2.2.4 concurrent_queue() [4/10]

```
template<class T , template< class, class > class Queue = std::deque, class Alloc = std::allocator<↩
T>>
concurrent_queue< T, Queue, Alloc >::concurrent_queue (
            size_type n,
            const value_type & val,
            const allocator_type & alloc = allocator_type() )  [inline]
```

Fill Constructor.

Constructs a container with n elements. Each element is a copy of val (if provided).

Parameters

in	*n*	number of elements
in	*val*	value to fill the concurrent queue with
in	*alloc*	optional memory allocator.

4.2.2.5 concurrent_queue() [5/10]

```
template<class T , template< class, class > class Queue = std::deque, class Alloc = std::allocator<↩
T>>
template<typename InputIterator >
concurrent_queue< T, Queue, Alloc >::concurrent_queue (
            InputIterator first,
            InputIterator last,
            const allocator_type & alloc = allocator_type() )  [inline]
```

Range Constructor.

Constructs a container with as many elements as the range [first,last), with each element emplace-constructed from its corresponding element in that range, in the same order.

Template Parameters

InputIterator	Input Iterator to value_type

Parameters

in	*first*	Iterator to the first element of the range
in	*last*	Iterator to the one-past-last element of the range

4.2.2.6 concurrent_queue() [6/10]

```
template<class T , template< class, class > class Queue = std::deque, class Alloc = std::allocator<↩
T>>
concurrent_queue< T, Queue, Alloc >::concurrent_queue (
            const concurrent_queue< T, Queue, Alloc > & other )  [inline]
```

Copy Constructor.

Constructs a container with a copy of each of the elements in x, in the same order. Thread-safe.

Parameters

in	*other*	queue to copy

4.2.2.7 concurrent_queue() [7/10]

```
template<class T , template< class, class > class Queue = std::deque, class Alloc = std::allocator<↩
T>>
concurrent_queue< T, Queue, Alloc >::concurrent_queue (
            const concurrent_queue< T, Queue, Alloc > & other,
            const allocator_type & alloc )  [inline]
```

Copy Constructor with allocator.

Constructs a container with a copy of each of the elements in x, in the same order. Thread-safe.

Parameters

in	*other*	queue to copy
in	*alloc*	optional memory allocator.

4.2.2.8 concurrent_queue() [8/10]

```
template<class T , template< class, class > class Queue = std::deque, class Alloc = std::allocator<↩
```

```
T>>
concurrent_queue< T, Queue, Alloc >::concurrent_queue (
            concurrent_queue< T, Queue, Alloc > && other )  [inline], [noexcept]
```

Move Constructor.

Constructs a container that acquires the elements of `other`. Ownership of the contained elements is directly transferred. `other` is left in an unspecified but valid state.

Parameters

in	*other*	container to move from

4.2.2.9 concurrent_queue() [9/10]

```
template<class T , template< class, class > class Queue = std::deque, class Alloc = std::allocator<↩
T>>
concurrent_queue< T, Queue, Alloc >::concurrent_queue (
            concurrent_queue< T, Queue, Alloc > && other,
            const allocator_type & alloc )  [inline], [noexcept]
```

Move Constructor.

Constructs a container that acquires the elements of `other`. Ownership of the contained elements is directly transferred. `other` is left in an unspecified but valid state.

Parameters

in	*other*	container to move from

4.2.2.10 concurrent_queue() [10/10]

```
template<class T , template< class, class > class Queue = std::deque, class Alloc = std::allocator<↩
T>>
concurrent_queue< T, Queue, Alloc >::concurrent_queue (
            std::initializer_list< T > init,
            const allocator_type & alloc = allocator_type() )  [inline]
```

Initializer List Constructor.

Parameters

in	*init*	initializer list to initialize the elements of the container with
in	*alloc*	allocator to use for all memory allocations of this container

4.2.2.11 ~concurrent_queue()

```
template<class T , template< class, class > class Queue = std::deque, class Alloc = std::allocator<↩
T>>
concurrent_queue< T, Queue, Alloc >::~concurrent_queue ( )  [inline], [default]
```

Destructor.

Destructs the concurrent queue. The destructors of the elements are called and the used storage is deallocated. Note, that if the elements are pointers, the pointed-to objects are not destroyed.

4.2.3 Member Function Documentation

4.2.3.1 acquire_read_lock()

```
template<class T , template< class, class > class Queue = std::deque, class Alloc = std::allocator<↩
T>>
read_lock_type concurrent_queue< T, Queue, Alloc >::acquire_read_lock ( ) const  [inline]
```

Lock the queue in a manner in which it is safe for multiple threads to read-iterate over it.

Allows concurrency-safe iteration.

Remarks

Intended for test and debug use only.

Returns

RAII lock suitable for safe read-iteration. Note that failure to move from or bind to the return value will invoke the locks destructor at the end of the call, instantly unlocking the concurrent queue.

4.2.3.2 acquire_write_lock()

```
template<class T , template< class, class > class Queue = std::deque, class Alloc = std::allocator<↩
T>>
write_lock_type concurrent_queue< T, Queue, Alloc >::acquire_write_lock ( )  [inline]
```

Lock the queue in a manner in which it is safe for multiple threads to write-iterate over it.

Allows concurrency-safe iteration.

Remarks

Intended for test and debug use only.

Returns

RAII lock suitable for safe write-iteration. Note that failure to move from or bind to the return value will invoke the locks destructor at the end of the call, instantly unlocking the concurrent queue.

4.2.3.3 begin() [1/2]

```
template<class T , template< class, class > class Queue = std::deque, class Alloc = std::allocator<↩
T>>
iterator concurrent_queue< T, Queue, Alloc >::begin ( )  [inline]
```

Returns an iterator of type iterator to the beginning of the concurrent queue.

This method is not concurrency-safe. The iterators for the concurrent_queue class are primarily intended for debugging, as they are slow, and iteration is not concurrency-safe with respect to other queue operations.

Returns

An iterator of type iterator to the beginning of the concurrent queue.

4.2.3.4 begin() [2/2]

```
template<class T , template< class, class > class Queue = std::deque, class Alloc = std::allocator<↩
T>>
const_iterator concurrent_queue< T, Queue, Alloc >::begin ( ) const  [inline]
```

Returns an iterator of type const_iterator to the beginning of the concurrent queue.

This method is not concurrency-safe. The iterators for the concurrent_queue class are primarily intended for debugging, as they are slow, and iteration is not concurrency-safe with respect to other queue operations.

Returns

An iterator of type const_iterator to the beginning of the concurrent queue.

4.2.3.5 cbegin()

```
template<class T , template< class, class > class Queue = std::deque, class Alloc = std::allocator<↩
T>>
const_iterator concurrent_queue< T, Queue, Alloc >::cbegin ( ) const  [inline]
```

Returns an iterator of type const_iterator to the beginning of the concurrent queue.

This method is not concurrency-safe. The iterators for the concurrent_queue class are primarily intended for debugging, as they are slow, and iteration is not concurrency-safe with respect to other queue operations.

Returns

An iterator of type const_iterator to the beginning of the concurrent queue.

4.2.3.6 cend()

```
template<class T , template< class, class > class Queue = std::deque, class Alloc = std::allocator<↩
T>>
const_iterator concurrent_queue< T, Queue, Alloc >::cend ( ) const  [inline]
```

Returns an iterator of type const_iterator to the end of the concurrent queue.

This method is not concurrency-safe. The iterators for the concurrent_queue class are primarily intended for debugging, as they are slow, and iteration is not concurrency-safe with respect to other queue operations.

Returns

An iterator of type const_iterator to the end of the concurrent queue.

4.2.3.7 clear()

```
template<class T , template< class, class > class Queue = std::deque, class Alloc = std::allocator<↩
T>>
void concurrent_queue< T, Queue, Alloc >::clear ( )  [inline], [noexcept]
```

Clears the concurrent queue, destroying any currently enqueued elements.

This method is not concurrency-safe, and invalidates all iterators.

4.2.3.8 emplace()

```
template<class T , template< class, class > class Queue = std::deque, class Alloc = std::allocator<↩
T>>
template<class... Args>
void concurrent_queue< T, Queue, Alloc >::emplace (
            Args &&... args )  [inline]
```

Constructs a new element in place at the end of the concurrent queue.

This method is concurrency-safe. This method is concurrency-safe with respect to calls to the methods `push`, `emplace`, `pop`, and `empty`.

Template Parameters

Args	Types of args, generally deduced automatically.

Parameters

in	*args*	Arguments to forward to the constructor of the element.

4.2.3.9 empty()

```
template<class T , template< class, class > class Queue = std::deque, class Alloc = std::allocator<↩
T>>
bool concurrent_queue< T, Queue, Alloc >::empty ( ) const  [inline], [noexcept]
```

Tests if the concurrent queue is empty at the moment this method is called.

This method is concurrency-safe. While this method is concurrency-safe with respect to calls to the methods `push`, `emplace`, `pop`, and `empty`, the value returned might be incorrect by the time it is inspected by the calling thread.

Returns

true if the concurrent queue was empty at the moment we looked, false otherwise.

4.2.3.10 end() [1/2]

```
template<class T , template< class, class > class Queue = std::deque, class Alloc = std::allocator<↩
T>>
iterator concurrent_queue< T, Queue, Alloc >::end ( )  [inline]
```

Returns an iterator of type iterator to the end of the concurrent queue.

This method is not concurrency-safe. The iterators for the concurrent_queue class are primarily intended for debugging, as they are slow, and iteration is not concurrency-safe with respect to other queue operations.

Returns

An iterator of type iterator to the end of the concurrent queue.

4.2.3.11 end() [2/2]

```
template<class T , template< class, class > class Queue = std::deque, class Alloc = std::allocator<↩
T>>
const_iterator concurrent_queue< T, Queue, Alloc >::end ( ) const  [inline]
```

Returns an iterator of type const_iterator to the end of the concurrent queue.

This method is not concurrency-safe. The iterators for the concurrent_queue class are primarily intended for debugging, as they are slow, and iteration is not concurrency-safe with respect to other queue operations.

Returns

An iterator of type const_iterator to the end of the concurrent queue.

4.2.3.12 get_allocator()

```
template<class T , template< class, class > class Queue = std::deque, class Alloc = std::allocator<↩
T>>
allocator_type concurrent_queue< T, Queue, Alloc >::get_allocator ( ) const  [inline], [noexcept]
```

Returns a copy of the allocator used to construct the concurrent queue.

This method is concurrency-safe.

Returns

A copy of the allocator used to construct the concurrent queue.

4.2.3.13 operator=() [1/2]

```
template<class T , template< class, class > class Queue = std::deque, class Alloc = std::allocator<↩
T>>
concurrent_queue& concurrent_queue< T, Queue, Alloc >::operator= (
            concurrent_queue< T, Queue, Alloc > && other )  [inline], [noexcept]
```

Move Assignment Operator.

Parameters

in	*other*	concurrent queue whose elements are to be moved.

Returns

A reference to this concurrent queue.

4.2.3.14 operator=() [2/2]

```
template<class T , template< class, class > class Queue = std::deque, class Alloc = std::allocator<↩
T>>
concurrent_queue& concurrent_queue< T, Queue, Alloc >::operator= (
            const concurrent_queue< T, Queue, Alloc > & other )  [inline]
```

Copy Assignment Operator.

Parameters

in	*other*	concurrent queue whose elements are to be copied.

Returns

A reference to this concurrent queue.

4.2.3.15 push() [1/2]

```
template<class T , template< class, class > class Queue = std::deque, class Alloc = std::allocator<↩
T>>
void concurrent_queue< T, Queue, Alloc >::push (
            const T & value )  [inline]
```

Enqueues an item at tail end of the concurrent queue.

This method is concurrency-safe. `push` is concurrency-safe with respect to calls to the methods `push` `emplace`, `pop`, and `empty`.

Parameters

in	*value*	The item to be added to the queue.

4.2.3.16 push() [2/2]

```
template<class T , template< class, class > class Queue = std::deque, class Alloc = std::allocator<↩
T>>
```

```
void concurrent_queue< T, Queue, Alloc >::push (
            T && value )  [inline]
```

Enqueues an item at tail end of the concurrent queue.

This method is concurrency-safe. `push` is concurrency-safe with respect to calls to the methods `push` `emplace`, `pop`, and `empty`.

Parameters

`in`	*value*	The item to be added to the queue.

4.2.3.17 size()

```
template<class T , template< class, class > class Queue = std::deque, class Alloc = std::allocator<↩
T>>
size_t concurrent_queue< T, Queue, Alloc >::size ( ) const  [inline]
```

Returns the number of items in the queue.

This method is concurrency-safe. `push` is concurrency-safe with respect to calls to the methods `push` `emplace`, `try_pop`, and `empty`.

Remarks

While calls to size are concurrency-safe in that they cannot damage the internal state of the concurrent queue, it is unwise to use the results as a condition of a `for` loop or for iteration, because the size of the container could change between the call to `size` and the invocation of the loop's methods.

Returns

The size of the concurrent queue.

4.2.3.18 try_pop()

```
template<class T , template< class, class > class Queue = std::deque, class Alloc = std::allocator<↩
T>>
bool concurrent_queue< T, Queue, Alloc >::try_pop (
            T & destination )  [inline]
```

Dequeues an item from the queue if one is available.

This method is concurrency-safe. If an item was successfully dequeued, the parameter `destination` receives the dequeued value, the original value held in the queue is destroyed, and this function returns true. If there was no item to dequeue, this function returns false without blocking, and the contents of the `destination` parameter are undefined. A false return value does not necessarily mean the queue is empty. `try_pop` is concurrency-safe with respect to calls to the methods `emplace`, `push`, `try_pop`, and `empty`.

Parameters

out	*destination*	A reference to a location to store the dequeued item.

Returns

true if an item was successfully dequeued, false otherwise.

4.2.3.19 try_pop_for()

```
template<class T , template< class, class > class Queue = std::deque, class Alloc = std::allocator<↩
T>>
template<class Rep , class Period >
bool concurrent_queue< T, Queue, Alloc >::try_pop_for (
            T & destination,
            const std::chrono::duration< Rep, Period > & timeout_duration )  [inline]
```

Dequeues an item from the queue if one is available within the specified timeout.

This method is concurrency-safe. If an item was successfully dequeued, the parameter `destination` receives the dequeued value, the original value held in the queue is destroyed, and this function returns true. If there was no item to dequeue, this function returns false without blocking, and the contents of the `destination` parameter are undefined. A false return value does not necessarily mean the queue is empty. `try_pop` is concurrency-safe with respect to calls to the methods `emplace`, `push`, `try_pop`, and `empty`.

Parameters

out	*destination*	A reference to a location to store the dequeued item.
in	*timeout_duration*	The maximum length of time `try_pop_for` will attempt to pop the queue before declaring failure.

Returns

true if an item was successfully dequeued, false otherwise.

4.2.4 Friends And Related Function Documentation

4.2.4.1 operator"!=

```
template<class T , template< class, class > class Queue = std::deque, class Alloc = std::allocator<↩
T>>
```

```
template<class Ty , template< class, class > class Q, class A >
bool operator!= (
            const concurrent_queue< Ty, Q, A > & lhs,
            const concurrent_queue< Ty, Q, A > & rhs )  [friend]
```

Inequality Operator.

Parameters

rhs	Right-hand side of the comparison

Returns

false if the contents of the queues are equivalent, true otherwise.

4.2.4.2 operator==

```
template<class T , template< class, class > class Queue = std::deque, class Alloc = std::allocator<↩
T>>
template<class Ty , template< class, class > class Q, class A >
bool operator== (
            const concurrent_queue< Ty, Q, A > & lhs,
            const concurrent_queue< Ty, Q, A > & rhs )  [friend]
```

Equality Operator.

Parameters

rhs	Right-hand side of the comparison

Returns

true if the contents of the queues are equivalent, false otherwise.

4.2.4.3 std::swap

```
template<class T , template< class, class > class Queue = std::deque, class Alloc = std::allocator<↩
T>>
template<class Ty , template< class, class > class Q, class A >
void std::swap (
            concurrent_queue< Ty, Q, A > & lhs,
            concurrent_queue< Ty, Q, A > & rhs )  [friend]
```

Swap Operator.

Parameters

lhs	container to swap with rhs
rhs	container to swap with lhs

The documentation for this class was generated from the following file:

- include/concurrent_queue.h

4.3 circular_queue< T, Alloc >::const_iterator Class Reference

constant iterator for the circular_queue class.

`#include <circular_queue.h>`

Public Types

- using **iterator_category** = std::random_access_iterator_tag
- using **value_type** = const T
- using **difference_type** = ptrdiff_t
- using **pointer** = const T *
- using **reference** = const T &

Public Member Functions

- const_iterator (const circular_queue *buffer, pointer start, bool parity) noexcept

  Constructor.
- **const_iterator** (const circular_queue *buffer, queue_pointer pointer) noexcept
- **const_iterator** (const const_iterator &other)=default
- const_iterator & **operator=** (const const_iterator &other)=default
- **const_iterator** (const_iterator &&other) noexcept
- const_iterator & **operator=** (const_iterator &&other) noexcept
- ~const_iterator () noexcept=default

  destructor. ***Complexity:*** *Constant.*
  Exception Safety: *No-throw guarantee: This function never throws exceptions.*
- reference operator* () const

  Dereference operator.
- pointer operator& () const

  Dereference operator.
- pointer operator-> () const

  Dereference operator.
- reference operator[ ] (difference_type n) const

  Dereference iterator with offset.

- bool operator== (const const_iterator &rhs) const noexcept

  equality operator
- bool operator!= (const const_iterator &rhs) const

  inequality operator ***Complexity:*** *Constant.*
  Iterator Validity: *No changes.*
  Data Races: *The container is accessed.*
  Exception Safety: *No-throw guarantee: This function never throws exceptions.*
- const_iterator & operator++ () noexcept

  Increment iterator position (prefix)
- const_iterator operator++ (int)

  Increment iterator position (postfix)
- const_iterator & operator+= (difference_type n) noexcept

  Advance iterator.
- const_iterator operator+ (difference_type n) const noexcept

  Addition operator.
- const_iterator & operator-- () noexcept

  Decrease iterator position (prefix)
- const_iterator operator-- (int)

  Decrease iterator position (postfix)
- const_iterator & operator-= (difference_type n) noexcept

  Retrocede iterator.
- const_iterator operator- (difference_type n) const

  subtraction operator
- difference_type operator- (const const_iterator &other) const

  subtraction operator
- bool operator< (const const_iterator &rhs) const noexcept

  less-than operator
- bool operator<= (const const_iterator &rhs) const

  less-than-or-equal operator
- bool operator> (const const_iterator &rhs) const

  greater-than operator
- bool operator>= (const const_iterator &rhs) const

  greater-than-or-equal operator

Protected Attributes

- const circular_queue * m_buffer = nullptr

  pointer to the buffer object that this iterates on.
- queue_pointer m_pointer

  Iterator position.

Friends

- class **circular_queue**

4.3.1 Detailed Description

template<class T, class Alloc = std::allocator<T>>
class circular_queue< T, Alloc >::const_iterator

constant iterator for the circular_queue class.

4.3.2 Constructor & Destructor Documentation

4.3.2.1 const_iterator()

```
template<class T , class Alloc = std::allocator<T>>
circular_queue< T, Alloc >::const_iterator::const_iterator (
            const circular_queue * buffer,
            pointer start,
            bool parity )  [inline], [noexcept]
```

Constructor.

Creates an initialized iterator. **Complexity:** Constant.
Iterator Validity: N/A.
Data Races: N/A.
Exception Safety: No-throw guarantee: This function never throws exceptions.

Parameters

in	*buffer*	pointer to the underlying container.
in	*start*	pointer to the iterators starting offset

4.3.3 Member Function Documentation

4.3.3.1 operator"!=()

```
template<class T , class Alloc = std::allocator<T>>
bool circular_queue< T, Alloc >::const_iterator::operator!= (
            const const_iterator & rhs ) const  [inline]
```

inequality operator **Complexity:** Constant.
Iterator Validity: No changes.

Data Races: The container is accessed.
Exception Safety: No-throw guarantee: This function never throws exceptions.

Parameters

in	*rhs*	right-hand-side of the equation.

Returns

true if both iterators point to different objects.

4.3.3.2 operator&()

```
template<class T , class Alloc = std::allocator<T>>
pointer circular_queue< T, Alloc >::const_iterator::operator& ( ) const  [inline]
```

Dereference operator.

Returns a pointer to the element pointed to by the iterator. **Complexity:** Constant.
Iterator Validity: Unchanged.
Data Races: The object is accessed.
Exception Safety: Undefined if the iterator is not valid.

Returns

A pointer to the element pointed by the iterator.

4.3.3.3 operator*()

```
template<class T , class Alloc = std::allocator<T>>
reference circular_queue< T, Alloc >::const_iterator::operator* ( ) const  [inline]
```

Dereference operator.

Returns a reference to the element pointed to by the iterator. **Complexity:** Constant.
Iterator Validity: No changes.
Data Races: The object is accessed.
Exception Safety: Undefined if the iterator is not valid.

Returns

A reference to the element pointed by the iterator.

4.3.3.4 operator+()

```
template<class T , class Alloc = std::allocator<T>>
const_iterator circular_queue< T, Alloc >::const_iterator::operator+ (
            difference_type n ) const  [inline], [noexcept]
```

Addition operator.

Returns an iterator pointing to the element located n positions away from the element the iterator currently points to.
Complexity: Constant.
Iterator Validity: Undefined behavior if the element *n* positions away is out of bounds.
Data Races: The object is accessed but NOT modified.
Exception Safety: Strong guarantee: if the constructor throws an exception, there are no side effects.

Parameters

in	*n*	number of elements to offset.

Returns

An iterator pointing to the element n positions away.

4.3.3.5 operator++() [1/2]

```
template<class T , class Alloc = std::allocator<T>>
const_iterator& circular_queue< T, Alloc >::const_iterator::operator++ ( )  [inline], [noexcept]
```

Increment iterator position (prefix)

Advances the iterator by 1 position. **Complexity:** Constant.
Iterator Validity: Valid IFF the iterator is incrementable.
Data Races: The object is modified.
Exception Safety: No-throw guarantee: This function never throws exceptions.

Returns

A reference to the incremented iterator.

4.3.3.6 operator++() [2/2]

```
template<class T , class Alloc = std::allocator<T>>
const_iterator circular_queue< T, Alloc >::const_iterator::operator++ (
            int  )  [inline]
```

Increment iterator position (postfix)

Advances the iterator by 1 position. **Complexity:** Constant.
Iterator Validity: Valid IFF the iterator is incrementable.
Data Races: The object is modified.
Exception Safety: Strong guarantee: if the function throws an exception, there are no side effects.

Returns

A copy of the iterator before it was incremented.

4.3.3.7 operator+=()

```
template<class T , class Alloc = std::allocator<T>>
const_iterator& circular_queue< T, Alloc >::const_iterator::operator+= (
            difference_type n )  [inline], [noexcept]
```

Advance iterator.

Advances the iterator by n element positions.
Complexity: Constant.
Iterator Validity: Results in undefined behavior if the element at position *n* does not exist.
Data Races: The object is modified.
Exception Safety: No-throw guarantee: This function never throws exceptions.

Parameters

[]	n

Returns

circular_queueIterator&

4.3.3.8 operator-() [1/2]

```
template<class T , class Alloc = std::allocator<T>>
difference_type circular_queue< T, Alloc >::const_iterator::operator- (
            const const_iterator & other ) const  [inline]
```

subtraction operator

Returns the distance between two iterators. **Complexity:** Constant.
Iterator Validity: Undefined behavior if the iterator is not decrementable, otherwise no changes.
Data Races: the object is accessed but NOT modified.
Exception Safety: Strong guarantee: if the constructor throws an exception, there are no side effects.

Parameters

in	*other*	iterator to determine distance from.

Returns

number of elements between the two iterators.

4.3.3.9 operator-() [2/2]

```
template<class T , class Alloc = std::allocator<T>>
const_iterator circular_queue< T, Alloc >::const_iterator::operator- (
            difference_type n ) const  [inline]
```

subtraction operator

Returns an iterator whose position is n elements before the current position **Complexity:** Constant.
Iterator Validity: Undefined behavior if the iterator is not decrementable, otherwise no changes.
Data Races: the object is accessed but NOT modified.
Exception Safety: Strong guarantee: if the constructor throws an exception, there are no side effects.

Parameters

in	*nNumber*	of elements to offset. Member type difference_type is an alias of the base container's own difference type.

Returns

An iterator decremented n positions from the current iterator position.

4.3.3.10 operator--() [1/2]

```
template<class T , class Alloc = std::allocator<T>>
const_iterator& circular_queue< T, Alloc >::const_iterator::operator-- ( )  [inline], [noexcept]
```

Decrease iterator position (prefix)

Decreases the iterator by one position.
Complexity: Constant.
Iterator Validity: Undefined behavior if the iterator is not decrementable, otherwise no changes.
Data Races: The object is modified.
Exception Safety: No-throw guarantee: This function never throws exceptions.

Returns

A reference to an iterator pointing to the post-decrement iterator.

4.3.3.11 operator--() [2/2]

```
template<class T , class Alloc = std::allocator<T>>
const_iterator circular_queue< T, Alloc >::const_iterator::operator-- (
            int  )  [inline]
```

Decrease iterator position (postfix)

Decreases the iterator by one position.
Complexity: Constant.
Iterator Validity: Undefined behavior if the iterator is not decrementable, otherwise no changes.
Data Races: The object is accessed but NOT modified.
Exception Safety: Strong guarantee: if the constructor throws an exception, there are no side effects.

Returns

A iterator pointing to the pre-decremented element.

4.3.3.12 operator-=()

```
template<class T , class Alloc = std::allocator<T>>
const_iterator& circular_queue< T, Alloc >::const_iterator::operator-= (
            difference_type n )  [inline], [noexcept]
```

Retrocede iterator.

Decreases the iterator by n element positions. **Complexity:** Constant.
Iterator Validity: Undefined behavior if the iterator is not decrementable, otherwise no changes.
Data Races: The object is modified.
Exception Safety: No-throw guarantee: This function never throws exceptions.

Parameters

in	*n*	Number of elements to offset. Member type difference_type is an alias of the base container's own difference type.

Returns

the iterator itself, decremented by n positions.

4.3.3.13 operator->()

```
template<class T , class Alloc = std::allocator<T>>
pointer circular_queue< T, Alloc >::const_iterator::operator-> ( ) const  [inline]
```

Dereference operator.

Returns a pointer to the element pointed to by the iterator. **Complexity:** Constant.
Iterator Validity: Unchanged.
Data Races: The object is accessed.
Exception Safety: Undefined if the iterator is not valid.

Returns

A pointer to the element pointed by the iterator.

4.3.3.14 operator<()

```
template<class T , class Alloc = std::allocator<T>>
bool circular_queue< T, Alloc >::const_iterator::operator< (
            const const_iterator & rhs ) const  [inline], [noexcept]
```

less-than operator

Performs the appropriate comparison. **Complexity:** Constant.
Iterator Validity: No changes.
Data Races: The object is accessed.
Exception Safety: No-throw guarantee: This function never throws exceptions.

Parameters

in	*rhs*	iterator for the right-hand side of the comparison.

Returns

true if this iterator is less than *rhs*.

4.3.3.15 operator<=()

```
template<class T , class Alloc = std::allocator<T>>
bool circular_queue< T, Alloc >::const_iterator::operator<= (
            const const_iterator & rhs ) const  [inline]
```

less-than-or-equal operator

Performs the appropriate comparison. **Complexity:** Constant.
Iterator Validity: No changes.
Data Races: The object is accessed.
Exception Safety: No-throw guarantee: This function never throws exceptions.

Parameters

in	*rhs*	iterator for the right-hand side of the comparison.

Returns

true if this iterator is less than or equal to *rhs*.

4.3.3.16 operator==()

```
template<class T , class Alloc = std::allocator<T>>
bool circular_queue< T, Alloc >::const_iterator::operator== (
            const const_iterator & rhs ) const  [inline], [noexcept]
```

equality operator

Two iterators are equal if they point to the same object and have the same parity value (i.e. one isn't wrapped around the buffer from the other).
Complexity: Constant.
Iterator Validity: No changes.
Data Races: The container is accessed.
Exception Safety: No-throw guarantee: This function never throws exceptions.

Parameters

in	*rhs*	right-hand-side of the equation.

Returns

true if both iterators point to the same object.

4.3.3.17 operator>()

```
template<class T , class Alloc = std::allocator<T>>
bool circular_queue< T, Alloc >::const_iterator::operator> (
            const const_iterator & rhs ) const  [inline]
```

greater-than operator

Performs the appropriate comparison. **Complexity:** Constant.
Iterator Validity: No changes.
Data Races: The object is accessed.
Exception Safety: No-throw guarantee: This function never throws exceptions.

Parameters

in	*rhs*	iterator for the right-hand side of the comparison.

Returns

true if this iterator is greater than *rhs*.

4.3.3.18 operator>=()

```
template<class T , class Alloc = std::allocator<T>>
bool circular_queue< T, Alloc >::const_iterator::operator>= (
            const const_iterator & rhs ) const  [inline]
```

greater-than-or-equal operator

Performs the appropriate comparison. **Complexity:** Constant.
Iterator Validity: No changes.
Data Races: The object is accessed.
Exception Safety: No-throw guarantee: This function never throws exceptions.

Parameters

in	*rhs*	iterator for the right-hand side of the comparison.

Returns

true if this iterator is greater than or equal to *rhs*.

4.3.3.19 operator[]()

```
template<class T , class Alloc = std::allocator<T>>
reference circular_queue< T, Alloc >::const_iterator::operator[] (
            difference_type n ) const  [inline]
```

Dereference iterator with offset.

Accesses the element located n positions away from the element currently pointed to by the iterator. If such an element does not exist, it causes *undefined behavior*. **Complexity:** Constant.
Iterator Validity: Unchanged.
Data Races: The object is accessed. Depending on the return type, the value returned may be used to access or modify elements.
Exception Safety: Undefined behavior if *n* is out of range.

Parameters

in	*n*	Number of elements to offset. Member type difference_type is an alias of the base container's own difference type.

Returns

The element n positions away from the element currently pointed by the iterator.

The documentation for this class was generated from the following file:

- include/circular_queue.h

4.4 circular_queue< T, Alloc >::const_queue_pointer Struct Reference

constant pointer with parity

```
#include <circular_queue.h>
```

Public Member Functions

- **const_queue_pointer** (pointer p, bool par)

Public Attributes

- const_pointer **ptr**
- bool **parity**

4.4.1 Detailed Description

template<class T, class Alloc = std::allocator<T>>
struct circular_queue< T, Alloc >::const_queue_pointer

constant pointer with parity

The documentation for this struct was generated from the following file:

- include/circular_queue.h

4.5 circular_queue< T, Alloc >::iterator Class Reference

iterator for the circular_queue class.

```
#include <circular_queue.h>
```

Public Types

- using **iterator_category** = std::random_access_iterator_tag
- using **value_type** = T
- using **difference_type** = ptrdiff_t
- using **pointer** = T *
- using **reference** = T &

Public Member Functions

- iterator (const circular_queue *const buffer, pointer start, bool parity) noexcept

  Constructor.
- **iterator** (const circular_queue *const buffer, queue_pointer pointer) noexcept
- **iterator** (const iterator &other)=default
- iterator & **operator=** (const iterator &other)=default
- **iterator** (iterator &&other) noexcept
- iterator & **operator=** (iterator &&other) noexcept
- ~iterator () noexcept=default

  destructor. ***Complexity:*** *Constant.*
  Exception Safety: *No-throw guarantee: This function never throws exceptions.*
- reference operator* () const

  Dereference operator.
- pointer operator& () const

  Dereference operator.
- pointer operator-> () const

  Dereference operator.
- reference operator[ ] (difference_type n) const

  Dereference iterator with offset.
- bool operator== (const iterator &rhs) const noexcept

  equality operator
- bool operator!= (const iterator &rhs) const

  inequality operator ***Complexity:*** *Constant.*
  Iterator Validity: *No changes.*
  Data Races: *The container is accessed.*
  Exception Safety: *No-throw guarantee: This function never throws exceptions.*
- iterator & operator++ () noexcept

  Increment iterator position (prefix)
- iterator operator++ (int)

  Increment iterator position (postfix)
- iterator & operator+= (difference_type n) noexcept

  Advance iterator.
- iterator operator+ (difference_type n) const noexcept

  Addition operator.
- iterator & operator-- () noexcept

  Decrease iterator position (prefix)
- iterator operator-- (int)

  Decrease iterator position (postfix)
- iterator & operator-= (difference_type n) noexcept

  Retrocede iterator.
- iterator operator- (difference_type n) const

  subtraction operator
- difference_type operator- (const iterator &other) const

  subtraction operator
- bool operator< (const iterator &rhs) const noexcept

  less-than operator

- bool operator<= (const iterator &rhs) const

  less-than-or-equal operator
- bool operator> (const iterator &rhs) const

  greater-than operator
- bool operator>= (const iterator &rhs) const

  greater-than-or-equal operator
- operator const_iterator () const

  Allow implicit conversion from iterator to const_iterator as required by STL.

Protected Attributes

- const circular_queue * **m_buffer** = nullptr
- queue_pointer m_pointer

  Iterator position.

Friends

- class **circular_queue**

4.5.1 Detailed Description

template<class T, class Alloc = std::allocator<T>>
class circular_queue< T, Alloc >::iterator

iterator for the circular_queue class.

4.5.2 Constructor & Destructor Documentation

4.5.2.1 iterator()

```
template<class T , class Alloc = std::allocator<T>>
circular_queue< T, Alloc >::iterator::iterator (
            const circular_queue *const buffer,
            pointer start,
            bool parity )  [inline], [noexcept]
```

Constructor.

Creates an initialized iterator. **Complexity:** Constant.
Iterator Validity: N/A.
Data Races: N/A.
Exception Safety: No-throw guarantee: This function never throws exceptions.

Parameters

in	*buffer*	pointer to the underlying container.
in	*start*	pointer to the iterator's starting offset

4.5.3 Member Function Documentation

4.5.3.1 operator"!=()

```
template<class T , class Alloc = std::allocator<T>>
bool circular_queue< T, Alloc >::iterator::operator!= (
            const iterator & rhs ) const  [inline]
```

inequality operator **Complexity:** Constant.
Iterator Validity: No changes.
Data Races: The container is accessed.
Exception Safety: No-throw guarantee: This function never throws exceptions.

Parameters

in	*rhs*	right-hand-side of the equation.

Returns

true if both iterators point to different objects.

4.5.3.2 operator&()

```
template<class T , class Alloc = std::allocator<T>>
pointer circular_queue< T, Alloc >::iterator::operator& ( ) const  [inline]
```

Dereference operator.

Returns a pointer to the element pointed to by the iterator. **Complexity:** Constant.
Iterator Validity: Unchanged.
Data Races: The object is accessed.
Exception Safety: Undefined if the iterator is not valid.

Returns

A pointer to the element pointed by the iterator.

4.5.3.3 operator∗()

```
template<class T , class Alloc = std::allocator<T>>
reference circular_queue< T, Alloc >::iterator::operator* ( ) const  [inline]
```

Dereference operator.

Returns a reference to the element pointed to by the iterator. **Complexity:** Constant.
Iterator Validity: No changes.
Data Races: The object is accessed.
Exception Safety: Undefined if the iterator is not valid.

Returns

A reference to the element pointed by the iterator.

4.5.3.4 operator+()

```
template<class T , class Alloc = std::allocator<T>>
iterator circular_queue< T, Alloc >::iterator::operator+ (
            difference_type n ) const  [inline], [noexcept]
```

Addition operator.

Returns an iterator pointing to the element located n positions away from the element the iterator currently points to.
Complexity: Constant.
Iterator Validity: Undefined behavior if the element *n* positions away is out of bounds.
Data Races: The object is accessed but NOT modified.
Exception Safety: Strong guarantee: if the constructor throws an exception, there are no side effects.

Parameters

in	*n*	number of elements to offset.

Returns

An iterator pointing to the element n positions away.

4.5.3.5 operator++() [1/2]

```
template<class T , class Alloc = std::allocator<T>>
iterator& circular_queue< T, Alloc >::iterator::operator++ ( )  [inline], [noexcept]
```

Increment iterator position (prefix)

Advances the iterator by 1 position. **Complexity:** Constant.
Iterator Validity: Valid IFF the iterator is incrementable.
Data Races: The object is modified.
Exception Safety: No-throw guarantee: This function never throws exceptions.

Returns

A reference to the incremented iterator.

4.5.3.6 operator++() [2/2]

```
template<class T , class Alloc = std::allocator<T>>
iterator circular_queue< T, Alloc >::iterator::operator++ (
            int  )  [inline]
```

Increment iterator position (postfix)

Advances the iterator by 1 position. **Complexity:** Constant.
Iterator Validity: Valid IFF the iterator is incrementable.
Data Races: The object is modified.
Exception Safety: Strong guarantee: if the function throws an exception, there are no side effects.

Returns

A copy of the iterator before it was incremented.

4.5.3.7 operator+=()

```
template<class T , class Alloc = std::allocator<T>>
iterator& circular_queue< T, Alloc >::iterator::operator+= (
            difference_type n )  [inline], [noexcept]
```

Advance iterator.

Advances the iterator by n element positions.
Complexity: Constant.
Iterator Validity: Results in undefined behavior if the element at position *n* does not exist.
Data Races: The object is modified.
Exception Safety: No-throw guarantee: This function never throws exceptions.

Parameters

[]	n

Returns

circular_queueIterator&

4.5.3.8 operator-() [1/2]

```
template<class T , class Alloc = std::allocator<T>>
difference_type circular_queue< T, Alloc >::iterator::operator- (
            const iterator & other ) const  [inline]
```

subtraction operator

Returns the distance between two iterators. **Complexity:** Constant.
Iterator Validity: Undefined behavior if the iterator is not decrementable, otherwise no changes.
Data Races: the object is accessed but NOT modified.
Exception Safety: Strong guarantee: if the constructor throws an exception, there are no side effects.

Parameters

in	*other*	iterator to determine distance from.

Returns

number of elements between the two iterators.

4.5.3.9 operator-() [2/2]

```
template<class T , class Alloc = std::allocator<T>>
iterator circular_queue< T, Alloc >::iterator::operator- (
            difference_type n ) const  [inline]
```

subtraction operator

Returns an iterator whose position is n elements before the current position **Complexity:** Constant.
Iterator Validity: Undefined behavior if the iterator is not decrementable, otherwise no changes.
Data Races: the object is accessed but NOT modified.
Exception Safety: Strong guarantee: if the constructor throws an exception, there are no side effects.

Parameters

in	*nNumber*	of elements to offset. Member type difference_type is an alias of the base container's own difference type.

Returns

An iterator decremented n positions from the current iterator position.

4.5.3.10 operator--() [1/2]

```
template<class T , class Alloc = std::allocator<T>>
iterator& circular_queue< T, Alloc >::iterator::operator-- ( )  [inline], [noexcept]
```

Decrease iterator position (prefix)

Decreases the iterator by one position.
Complexity: Constant.
Iterator Validity: Undefined behavior if the iterator is not decrementable, otherwise no changes.
Data Races: The object is modified.
Exception Safety: No-throw guarantee: This function never throws exceptions.

Returns

A reference to an iterator pointing to the post-decrement iterator.

4.5.3.11 operator--() [2/2]

```
template<class T , class Alloc = std::allocator<T>>
iterator circular_queue< T, Alloc >::iterator::operator-- (
            int  )  [inline]
```

Decrease iterator position (postfix)

Decreases the iterator by one position.
Complexity: Constant.
Iterator Validity: Undefined behavior if the iterator is not decrementable, otherwise no changes.
Data Races: The object is accessed but NOT modified.
Exception Safety: Strong guarantee: if the constructor throws an exception, there are no side effects.

Returns

A iterator pointing to the pre-decremented element.

4.5.3.12 operator-=()

```
template<class T , class Alloc = std::allocator<T>>
iterator& circular_queue< T, Alloc >::iterator::operator-= (
            difference_type n )  [inline], [noexcept]
```

Retrocede iterator.

Decreases the iterator by n element positions. **Complexity:** Constant.
Iterator Validity: Undefined behavior if the iterator is not decrementable, otherwise no changes.
Data Races: The object is modified.
Exception Safety: No-throw guarantee: This function never throws exceptions.

Parameters

in	*n*	Number of elements to offset. Member type difference_type is an alias of the base container's own difference type.

Returns

the iterator itself, decremented by n positions.

4.5.3.13 operator->()

```
template<class T , class Alloc = std::allocator<T>>
pointer circular_queue< T, Alloc >::iterator::operator-> ( ) const  [inline]
```

Dereference operator.

Returns a pointer to the element pointed to by the iterator. **Complexity:** Constant.
Iterator Validity: Unchanged.
Data Races: The object is accessed.
Exception Safety: Undefined if the iterator is not valid.

Returns

A pointer to the element pointed by the iterator.

4.5.3.14 operator<()

```
template<class T , class Alloc = std::allocator<T>>
bool circular_queue< T, Alloc >::iterator::operator< (
            const iterator & rhs ) const  [inline], [noexcept]
```

less-than operator

Performs the appropriate comparison. **Complexity:** Constant.
Iterator Validity: No changes.
Data Races: The object is accessed.
Exception Safety: No-throw guarantee: This function never throws exceptions.

Parameters

in	*rhs*	iterator for the right-hand side of the comparison.

Returns

true if this iterator is less than *rhs*.

4.5.3.15 operator<=()

```
template<class T , class Alloc = std::allocator<T>>
bool circular_queue< T, Alloc >::iterator::operator<= (
            const iterator & rhs ) const  [inline]
```

less-than-or-equal operator

Performs the appropriate comparison. **Complexity:** Constant.
Iterator Validity: No changes.
Data Races: The object is accessed.
Exception Safety: No-throw guarantee: This function never throws exceptions.

Parameters

in	*rhs*	iterator for the right-hand side of the comparison.

Returns

true if this iterator is less than or equal to *rhs*.

4.5.3.16 operator==()

```
template<class T , class Alloc = std::allocator<T>>
bool circular_queue< T, Alloc >::iterator::operator== (
            const iterator & rhs ) const  [inline], [noexcept]
```

equality operator

Two iterators are equal if they point to the same object and have the same parity value (i.e. one isn't wrapped around the buffer from the other).
Complexity: Constant.
Iterator Validity: No changes.
Data Races: The container is accessed.
Exception Safety: No-throw guarantee: This function never throws exceptions.

Parameters

in	*rhs*	right-hand-side of the equation.

Returns

true if both iterators point to the same object.

4.5.3.17 operator>()

```
template<class T , class Alloc = std::allocator<T>>
bool circular_queue< T, Alloc >::iterator::operator> (
            const iterator & rhs ) const  [inline]
```

greater-than operator

Performs the appropriate comparison. **Complexity:** Constant.
Iterator Validity: No changes.
Data Races: The object is accessed.
Exception Safety: No-throw guarantee: This function never throws exceptions.

Parameters

in	*rhs*	iterator for the right-hand side of the comparison.

Returns

true if this iterator is greater than *rhs*.

4.5.3.18 operator>=()

```
template<class T , class Alloc = std::allocator<T>>
bool circular_queue< T, Alloc >::iterator::operator>= (
            const iterator & rhs ) const  [inline]
```

greater-than-or-equal operator

Performs the appropriate comparison. **Complexity:** Constant.
Iterator Validity: No changes.
Data Races: The object is accessed.
Exception Safety: No-throw guarantee: This function never throws exceptions.

Parameters

in	*rhs*	iterator for the right-hand side of the comparison.

Returns

true if this iterator is greater than or equal to *rhs*.

4.5.3.19 operator[]()

```
template<class T , class Alloc = std::allocator<T>>
reference circular_queue< T, Alloc >::iterator::operator[] (
            difference_type n ) const  [inline]
```

Dereference iterator with offset.

Accesses the element located n positions away from the element currently pointed to by the iterator. If such an element does not exist, it causes *undefined behavior*. **Complexity:** Constant.
Iterator Validity: Unchanged.
Data Races: The object is accessed. Depending on the return type, the value returned may be used to access or modify elements.
Exception Safety: Undefined behavior if *n* is out of range.

Parameters

in	*n*	Number of elements to offset. Member type difference_type is an alias of the base container's own difference type.

Returns

The element n positions away from the element currently pointed by the iterator.

The documentation for this class was generated from the following file:

- include/circular_queue.h

4.6 circular_queue< T, Alloc >::queue_pointer Struct Reference

pointer with parity

`#include <circular_queue.h>`

Public Member Functions

- **queue_pointer** (pointer p, bool par)

Public Attributes

- pointer **ptr**
- bool **parity**

4.6.1 Detailed Description

template<class T, class Alloc = std::allocator<T>>
struct circular_queue< T, Alloc >::queue_pointer

pointer with parity

The documentation for this struct was generated from the following file:

- include/circular_queue.h

Chapter 5

File Documentation

5.1 include/circular_queue.h File Reference

An STL-style fixed-size circular queue.

```
#include <algorithm>
#include <cassert>
#include <cstddef>
#include <cstring>
#include <initializer_list>
#include <iterator>
#include <stdexcept>
#include <cstdint>
#include <type_traits>
```

Classes

- class circular_queue< T, Alloc >
  Fixed capacity, STL-style, templated circular buffer.
- struct circular_queue< T, Alloc >::queue_pointer
  pointer with parity
- struct circular_queue< T, Alloc >::const_queue_pointer
  constant pointer with parity
- class circular_queue< T, Alloc >::const_iterator
  constant iterator for the circular_queue class.
- class circular_queue< T, Alloc >::iterator
  iterator for the circular_queue class.

5.1.1 Detailed Description

An STL-style fixed-size circular queue.

5.2 include/concurrent_queue.h File Reference

A thread-safe queue (FIFO) implementation.

```
#include <chrono>
#include <condition_variable>
#include <deque>
#include <memory>
#include <mutex>
#include <shared_mutex>
```

Classes

- class concurrent_queue< T, Queue, Alloc >

  The `concurrent_queue` class is a sequence container class that allows first-in, first-out access to its elements.

Functions

- template<class T , template< class, class > class Queue = std::deque, class Alloc = std::allocator<T>>
  bool operator== (const concurrent_queue< T, Queue, Alloc > &lhs, const concurrent_queue< T, Queue, Alloc > &rhs)

  Equality Operator.
- template<class T , template< class, class > class Queue = std::deque, class Alloc = std::allocator<T>>
  bool operator!= (const concurrent_queue< T, Queue, Alloc > &lhs, const concurrent_queue< T, Queue, Alloc > &rhs)

  Inequality Operator.
- template<class T , template< class, class > class Queue = std::deque, class Alloc = std::allocator<T>>
  void std::swap (concurrent_queue< T, Queue, Alloc > &lhs, concurrent_queue< T, Queue, Alloc > &rhs)

  Swap Operator.

5.2.1 Detailed Description

A thread-safe queue (FIFO) implementation.

5.2.2 Function Documentation

5.2.2.1 operator"!=()

```
template<class T , template< class, class > class Queue = std::deque, class Alloc = std::allocator<↩
T>>
bool operator!= (
            const concurrent_queue< T, Queue, Alloc > & lhs,
            const concurrent_queue< T, Queue, Alloc > & rhs )  [inline]
```

Inequality Operator.

Returns

true if the queues are not element-by-element equivalent

5.2.2.2 operator==()

```
template<class T , template< class, class > class Queue = std::deque, class Alloc = std::allocator<↩
T>>
bool operator== (
            const concurrent_queue< T, Queue, Alloc > & lhs,
            const concurrent_queue< T, Queue, Alloc > & rhs )  [inline]
```

Equality Operator.

Returns

true if the queues are element-by-element equivalent

Index