

# Package ‘cppcontainers’

January 15, 2025

**Type** Package

**Title** 'C++' Standard Template Library Containers

**Version** 1.0.4

**Description** Use 'C++' Standard Template Library containers interactively in R. Includes sets, unordered sets, multisets, unordered multisets, maps, unordered maps, multimaps, unordered multimaps, stacks, queues, priority queues, vectors, deques, forward lists, and lists.

**Encoding** UTF-8

**URL** <https://github.com/cdueben/cppcontainers>

**BugReports** <https://github.com/cdueben/cppcontainers/issues>

**License** MIT + file LICENSE

**Imports** base (>= 4.0.0), methods, Rcpp

**LinkingTo** Rcpp

**SystemRequirements** C++20

**RoxygenNote** 7.3.2

**Collate** 'RcppExports.R' 'utils.R' 'classes.R' 'assign.R' 'at.R' 'back.R' 'bucket\_count.R' 'capacity.R' 'clear.R' 'contains.R' 'count.R' 'deque.R' 'emplace.R' 'emplace\_after.R' 'emplace\_back.R' 'emplace\_front.R' 'empty.R' 'erase.R' 'erase\_after.R' 'flip.R' 'forward\_list.R' 'front.R' 'insert.R' 'insert\_after.R' 'insert\_or\_assign.R' 'list.R' 'load\_factor.R' 'map.R' 'max\_bucket\_count.R' 'max\_load\_factor.R' 'max\_size.R' 'merge.R' 'multimap.R' 'multiset.R' 'operators.R' 'pop.R' 'pop\_back.R' 'pop\_front.R' 'print.R' 'priority\_queue.R' 'push.R' 'push\_back.R' 'push\_front.R' 'queue.R' 'rehash.R' 'remove..R' 'reserve.R' 'resize.R' 'reverse.R' 'set.R' 'show.R' 'shrink\_to\_fit.R' 'size.R' 'sort.R' 'sorting.R' 'splice.R' 'splice\_after.R' 'stack.R' 'to\_r.R' 'top.R' 'try\_emplace.R' 'type.R' 'unique.R' 'unordered\_map.R' 'unordered\_multimap.R' 'unordered\_multiset.R' 'unordered\_set.R' 'vector.R'

**Suggests** knitr, rmarkdown, testthat (>= 3.0.0)

**VignetteBuilder** knitr

**Config/testthat/edition** 3

**NeedsCompilation** yes

**Author** Christian Dübén [aut, cre]

**Maintainer** Christian Dübén <cdueben.ml+cran@proton.me>

**Repository** CRAN

**Date/Publication** 2025-01-15 08:00:14 UTC

## Contents

==,CppSet,CppSet-method . . . . .	3
assign . . . . .	5
at . . . . .	6
back . . . . .	7
bucket_count . . . . .	8
capacity . . . . .	9
clear . . . . .	10
contains . . . . .	10
count . . . . .	11
cpp_deque . . . . .	12
cpp_forward_list . . . . .	13
cpp_list . . . . .	14
cpp_map . . . . .	15
cpp_multimap . . . . .	16
cpp_multiset . . . . .	17
cpp_priority_queue . . . . .	19
cpp_queue . . . . .	20
cpp_set . . . . .	21
cpp_stack . . . . .	22
cpp_unordered_map . . . . .	23
cpp_unordered_multimap . . . . .	24
cpp_unordered_multiset . . . . .	25
cpp_unordered_set . . . . .	27
cpp_vector . . . . .	28
emplace . . . . .	29
emplace_after . . . . .	30
emplace_back . . . . .	31
emplace_front . . . . .	32
empty . . . . .	33
erase . . . . .	34
erase_after . . . . .	35
flip . . . . .	36
front . . . . .	37
insert . . . . .	38
insert_after . . . . .	39
insert_or_assign . . . . .	40

load_factor . . . . .	41
max_bucket_count . . . . .	41
max_load_factor . . . . .	42
max_size . . . . .	43
merge . . . . .	44
pop . . . . .	45
pop_back . . . . .	46
pop_front . . . . .	47
print . . . . .	47
push . . . . .	49
push_back . . . . .	50
push_front . . . . .	50
rehash . . . . .	51
remove. . . . .	52
reserve . . . . .	53
resize . . . . .	54
reverse . . . . .	55
shrink_to_fit . . . . .	55
size . . . . .	56
sort . . . . .	57
sorting . . . . .	58
splice . . . . .	59
splice_after . . . . .	60
top . . . . .	61
to_r . . . . .	61
try_emplace . . . . .	63
type . . . . .	64
unique . . . . .	65
[, CppMap-method . . . . .	66

**Index** **68**

==, CppSet, CppSet-method  
*Check equality*

**Description**

Check, if two containers hold identical data.

**Usage**

```
## S4 method for signature 'CppSet, CppSet'
e1 == e2

## S4 method for signature 'CppUnorderedSet, CppUnorderedSet'
e1 == e2
```

```

## S4 method for signature 'CppMultiset, CppMultiset'
e1 == e2

## S4 method for signature 'CppUnorderedMultiset, CppUnorderedMultiset'
e1 == e2

## S4 method for signature 'CppMap, CppMap'
e1 == e2

## S4 method for signature 'CppUnorderedMap, CppUnorderedMap'
e1 == e2

## S4 method for signature 'CppMultimap, CppMultimap'
e1 == e2

## S4 method for signature 'CppUnorderedMultimap, CppUnorderedMultimap'
e1 == e2

## S4 method for signature 'CppStack, CppStack'
e1 == e2

## S4 method for signature 'CppQueue, CppQueue'
e1 == e2

## S4 method for signature 'CppVector, CppVector'
e1 == e2

## S4 method for signature 'CppDeque, CppDeque'
e1 == e2

## S4 method for signature 'CppForwardList, CppForwardList'
e1 == e2

## S4 method for signature 'CppList, CppList'
e1 == e2

```

**Arguments**

- e1            A CppSet, CppUnorderedSet, CppMultiset, CppUnorderedMultiset, CppMap, CppUnorderedMap, CppMultimap, CppUnorderedMultimap, CppStack, CppQueue, CppVector, CppDeque, CppForwardList, or CppList object.
- e2            A CppSet, CppUnorderedSet, CppMultiset, CppUnorderedMultiset, CppMap, CppUnorderedMap, CppMultimap, CppUnorderedMultimap, CppStack, CppQueue, CppVector, CppDeque, CppForwardList, or CppList object of the same class and data type as e1.

**Value**

Returns TRUE, if the containers hold the same data and FALSE otherwise.

**See Also**

[contains](#), [type](#), [sorting](#).

**Examples**

```
x <- cpp_set(1:10)
y <- cpp_set(1:10)
x == y
# [1] TRUE

y <- cpp_set(1:11)
x == y
# [1] FALSE
```

---

assign

*Replace all elements*

---

**Description**

Replace all elements in a container by reference.

**Usage**

```
assign(x, value, pos, envir, inherits, immediate)
```

**Arguments**

x	A CppVector, CppDeque, CppForwardList, or CppList object.
value	A vector. It is called value instead of values for compliance with the generic base::assign method.
pos	Ignored.
envir	Ignored.
inherits	Ignored.
immediate	Ignored.

**Details**

Replaces all elements in x with the elements in value.

The parameters pos, envir, inherits, and immediate are only included for compatibility with the generic base::assign method and have no effect.

**Value**

Invisibly returns NULL.

**See Also**

[emplace](#), [emplace\\_after](#), [emplace\\_back](#), [emplace\\_front](#), [insert](#), [insert\\_after](#), [insert\\_or\\_assign](#).

**Examples**

```
v <- cpp_vector(4:9)
v
# 4 5 6 7 8 9

assign(v, 12:14)
v
# 12 13 14
```

---

at *Access elements with bounds checking*

---

**Description**

Read a value at a certain position with bounds checking.

**Usage**

```
at(x, position)
```

**Arguments**

x	A CppMap, CppUnorderedMap, CppVector, or CppDeque object.
position	A key (CppMap, CppUnorderedMap) or index (CppVector, CppDeque).

**Details**

In the two associative container types (CppMap, CppUnorderedMap), [] accesses a value by its key. If the key does not exist, the function throws an error.

In the two sequence container types (CppVector, CppDeque), [] accesses a value by its index. If the index is outside the container, this throws an error.

[at](#) and [] both access elements. Unlike [], [at](#) checks the bounds of the container and throws an error, if the element does not exist.

**Value**

Returns the value at the position.

**See Also**

[\[\]](#), [back](#), [contains](#), [front](#), [top](#).

**Examples**

```
m <- cpp_map(4:6, seq.int(0, 1, by = 0.5))
m
# [4,0] [5,0.5] [6,1]

at(m, 4L)
# [1] 0

d <- cpp_deque(c("hello", "world"))
d
# "hello" "world"

at(d, 2)
# [1] "world"
```

---

back

*Access last element*

---

**Description**

Access the last element in a container without removing it.

**Usage**

```
back(x)
```

**Arguments**

x                    A CppQueue, CppVector, CppDeque, or CppList object.

**Details**

In a CppQueue, the last element is the last inserted one.

**Value**

Returns the last element.

**See Also**

[front](#), [top](#), [push\\_back](#), [emplace\\_back](#), [pop\\_back](#).

**Examples**

```
q <- cpp_queue(1:4)
q
# First element: 1

back(q)
# [1] 4

l <- cpp_list(1:4)
l
# 1 2 3 4

back(l)
# [1] 4
```

---

bucket\_count

*Get the number of buckets*

---

**Description**

Obtain the container's number of buckets.

**Usage**

```
bucket_count(x)
```

**Arguments**

x                    A CppUnorderedSet, CppUnorderedMultiset, CppUnorderedMap, CppUnordered-Multimap object.

**Value**

Returns the container's number of buckets.

**See Also**

[max\\_bucket\\_count](#), [load\\_factor](#), [size](#).

**Examples**

```
s <- cpp_unordered_set(6:10)
s
# 10 9 8 7 6

bucket_count(s)
# [1] 13
```



---

capacity	<i>Get container capacity</i>
----------	-------------------------------

---

### Description

Get the capacity of a CppVector.

### Usage

```
capacity(x)
```

### Arguments

x                    A CppVector object.

### Details

The capacity is the space reserved for the vector, which can exceed its [size](#). Additional capacity ensures that the vector can be extended efficiently, without having to reallocate its current elements to a new memory location.

### Value

Returns a numeric.

### See Also

[reserve](#), [shrink\\_to\\_fit](#), [size](#).

### Examples

```
v <- cpp_vector(4:9)
v
# 4 5 6 7 8 9

capacity(v)
# [1] 6

reserve(v, 20)
size(v)
#[1] 6
capacity(v)
# [1] 20

v
# 4 5 6 7 8 9
```

clear *Clear the container*

---

**Description**

Remove all elements from a container by reference.

**Usage**

```
clear(x)
```

**Arguments**

x A CppSet, CppUnorderedSet, CppMultiset, CppUnorderedMultiset, CppMap, CppUnorderedMap, CppMultimap, CppUnorderedMultimap, CppVector, CppDeque, CppForwardList, or CppList object.

**Value**

Invisibly returns NULL.

**See Also**

[erase](#), [remove.](#), [empty](#).

**Examples**

```
l <- cpp_forward_list(4:9)
l
# 4 5 6 7 8 9

clear(l)
l
#
empty(l)
# [1] TRUE
```

---

contains *Check for elements*

---

**Description**

Check, if elements are part of a container.

**Usage**

```
contains(x, values)
```

**Arguments**

x	A CppSet, CppUnorderedSet, CppMultiset, CppUnorderedMultiset, CppMap, CppUnorderedMap, CppMultimap, or CppUnorderedMultimap object.
values	Values whose existence to assess. Refers to keys in the case of CppMap, CppUnorderedMap, CppMultimap, and CppUnorderedMultimap objects.

**Value**

Returns a logical vector of the same length as values, denoting for each value whether it is part of x (TRUE) or not (FALSE).

**See Also**

[\[, at, back, front, top.](#)

**Examples**

```
s <- cpp_multiset(4:9)
s
# 4 5 6 7 8 9

contains(s, 9:11)
# [1] TRUE FALSE FALSE

m <- cpp_unordered_map(c("hello", "world"), 3:4)
m
# ["world",4] ["hello",3]

contains(m, c("world", "there"))
# [1] TRUE FALSE
```

---

count	<i>Count element frequency</i>
-------	--------------------------------

---

**Description**

Count how often elements occur in a container.

**Usage**

```
count(x, values)
```

**Arguments**

x	A CppSet, CppUnorderedSet, CppMultiset, CppUnorderedMultiset, CppMap, CppUnorderedMap, CppMultimap, or CppUnorderedMultimap object.
values	A vector of elements to check. Refers to keys in the case of CppMap, CppUnorderedMap, CppMultimap, and CppUnorderedMultimap objects.

**Value**

Returns a vector of the same length as values, denoting for each value how often it occurs.

**See Also**

[\[, ==, at, contains, size, empty](#).

**Examples**

```
s <- cpp_set(4:9)
s
# 4 5 6 7 8 9

count(s, 9:11)
# [1] 1 0 0

m <- cpp_map(c("hello", "there"), c(1.2, 1.3))
m
# ["hello",1.2] ["there",1.3]

count(m, c("hello", "world"))
# [1] 1 0
```

---

cpp\_deque

*Create deque*

---

**Description**

Create a deque, i.e. a double-ended queue.

**Usage**

```
cpp_deque(x)
```

**Arguments**

x                   An integer, numeric, character, or logical vector.

**Details**

C++ deque methods implemented in this package are [assign](#), [at](#), [back](#), [clear](#), [emplace](#), [emplace\\_back](#), [emplace\\_front](#), [empty](#), [erase](#), [front](#), [insert](#), [max\\_size](#), [pop\\_back](#), [pop\\_front](#), [push\\_back](#), [push\\_front](#), [resize](#), [shrink\\_to\\_fit](#), and [size](#). The package also adds the `==` and `[]` operators and various helper functions ([print](#), [to\\_r](#), [type](#)).

All object-creating methods in this package begin with `cpp_` to avoid clashes with functions from other packages, such as `utils::stack` and `base::vector`.

**Value**

Returns a CppDeque object referencing a deque in C++.

**See Also**

[cpp\\_vector](#), [cpp\\_forward\\_list](#), [cpp\\_list](#).

**Examples**

```
d <- cpp_deque(4:6)
d
# 4 5 6

push_back(d, 1L)
d
# 4 5 6 1

push_front(d, 2L)
d
# 2 4 5 6 1
```

---

cpp_forward_list	<i>Create forward list</i>
------------------	----------------------------

---

**Description**

Create a forward list, i.e. a singly-linked list.

**Usage**

```
cpp_forward_list(x)
```

**Arguments**

x                    An integer, numeric, character, or logical vector.

**Details**

@details Singly-linked means that list elements store a reference only to the following element. This container type, thus, requires less RAM than a doubly-linked list does, but can only be iterated in the forward direction.

C++ forward\_list methods implemented in this package are [assign](#), [clear](#), [emplace\\_after](#), [emplace\\_front](#), [empty](#), [erase\\_after](#), [front](#), [insert\\_after](#), [max\\_size](#), [pop\\_front](#), [push\\_front](#), [remove.](#), [resize](#), [reverse](#), [sort](#), [splice\\_after](#), and [unique](#). The package also adds the == operator and various helper functions ([print](#), [to\\_r](#), [type](#)).

All object-creating methods in this package begin with cpp\_ to avoid clashes with functions from other packages, such as `utils::stack` and `base::vector`.

**Value**

Returns a CppForwardList object referencing a forward\_list in C++.

**See Also**

[cpp\\_vector](#), [cpp\\_deque](#), [cpp\\_list](#).

**Examples**

```
v <- cpp_forward_list(4:6)
v
# 4 5 6

push_front(v, 10L)
v
# 10 4 5 6

pop_front(v)
v
# 4 5 6
```

---

cpp\_list

*Create list*


---

**Description**

Create a list, i.e. a doubly-linked list.

**Usage**

```
cpp_list(x)
```

**Arguments**

x                    An integer, numeric, character, or logical vector.

**Details**

Doubly-linked means that list elements store a reference both to the previous element and to the following element. This container type, thus, requires more RAM than a singly-linked list does, but can be iterated in both directions.

C++ list methods implemented in this package are [assign](#), [back](#), [clear](#), [emplace](#), [emplace\\_back](#), [emplace\\_front](#), [empty](#), [erase](#), [front](#), [insert](#), [max\\_size](#), [merge](#), [pop\\_back](#), [pop\\_front](#), [push\\_back](#), [push\\_front](#), [remove.](#), [resize](#), [reverse](#), [size](#), [sort](#), [splice](#), and [unique](#). The package also adds the `==` operator and various helper functions ([print](#), [to\\_r](#), [type](#)).

All object-creating methods in this package begin with `cpp_` to avoid clashes with functions from other packages, such as `utils::stack` and `base::vector`.

**Value**

Returns a CppList object referencing a list in C++.

**See Also**

[cpp\\_vector](#), [cpp\\_deque](#), [cpp\\_forward\\_list](#).

**Examples**

```
l <- cpp_list(4:6)
l
# 4 5 6

push_back(l, 1L)
l
# 4 5 6 1

push_front(l, 2L)
l
# 2 4 5 6 1
```

---

 cpp\_map

*Create map*


---

**Description**

Create a map. Maps are key-value pairs sorted by unique keys.

**Usage**

```
cpp_map(keys, values)
```

**Arguments**

keys	An integer, numeric, character, or logical vector.
values	An integer, numeric, character, or logical vector.

**Details**

Maps are associative containers. They do not provide random access through an index. I.e. `m[2]` does not return the second element.

C++ map methods implemented in this package are [at](#), [clear](#), [contains](#), [count](#), [emplace](#), [empty](#), [erase](#), [insert](#), [insert\\_or\\_assign](#), [max\\_size](#), [merge](#), [size](#), and [try\\_emplace](#). The package also adds the `==` and `[]` operators and various helper functions ([print](#), [to\\_r](#), [type](#)).

All object-creating methods in this package begin with `cpp_` to avoid clashes with functions from other packages, such as `utils::stack` and `base::vector`.

**Value**

Returns a CppMap object referencing a map in C++.

**See Also**

[cpp\\_unordered\\_map](#), [cpp\\_multimap](#), [cpp\\_unordered\\_multimap](#).

**Examples**

```
m <- cpp_map(4:6, seq.int(1, by = 0.5, length.out = 3L))
m
# [4,1] [5,1.5] [6,2]

insert(m, seq.int(100, by = 0.1, length.out = 3L), 14:16)
m
# [4,1] [5,1.5] [6,2] [14,100] [15,100.1] [16,100.2]

print(m, from = 6L)
# [6,2] [14,100] [15,100.1] [16,100.2]

m <- cpp_map(c("world", "hello", "there"), 4:6)
m
# ["hello",5] ["there",6] ["world",4]

erase(m, "there")
m
# ["hello",5] ["world",4]
```

---

cpp\_multimap

*Create multimap*

---

**Description**

Create a multimap. Multimaps are key-value pairs sorted by non-unique keys.

**Usage**

```
cpp_multimap(keys, values)
```

**Arguments**

keys	An integer, numeric, character, or logical vector.
values	An integer, numeric, character, or logical vector.



## Details

Multimaps are associative containers. They do not provide random access through an index. I.e. `m[2]` does not return the second element.

C++ multimap methods implemented in this package are [clear](#), [contains](#), [count](#), [emplace](#), [empty](#), [erase](#), [insert](#), [max\\_size](#), [merge](#), and [size](#). The package also adds the `==` operator and various helper functions ([print](#), [to\\_r](#), [type](#)).

All object-creating methods in this package begin with `cpp_` to avoid clashes with functions from other packages, such as `utils::stack` and `base::vector`.

## Value

Returns a `CppMultimap` object referencing a multimap in C++.

## See Also

[cpp\\_map](#), [cpp\\_unordered\\_map](#), [cpp\\_unordered\\_multimap](#).

## Examples

```
m <- cpp_multimap(4:6, seq.int(1, by = 0.5, length.out = 3L))
m
# [4,1] [5,1.5] [6,2]

insert(m, seq.int(100, by = 0.1, length.out = 3L), 5:7)
m
# [4,1] [5,1.5] [5,100] [6,2] [6,100.1] [7,100.2]

print(m, from = 6)
# [6,2] [6,100.1] [7,100.2]

m <- cpp_multimap(c("world", "hello", "there", "world"), 3:6)
m
# ["hello",4] ["there",5] ["world",3] ["world",6]

erase(m, "world")
m
# ["hello",4] ["there",5]
```

---

cpp\_multiset

*Create multiset*

---

## Description

Create a multiset. Multisets are containers of sorted, non-unique elements.

## Usage

```
cpp_multiset(x)
```

## Arguments

x                    An integer, numeric, character, or logical vector.

## Details

Multisets are associative containers. They do not provide random access through an index. I.e. `s[2]` does not return the second element.

C++ multiset methods implemented in this package are [clear](#), [contains](#), [count](#), [emplace](#), [empty](#), [erase](#), [insert](#), [max\\_size](#), [merge](#), and [size](#). The package also adds the `==` operator and various helper functions ([print](#), [to\\_r](#), [type](#)).

All object-creating methods in this package begin with `cpp_` to avoid clashes with functions from other packages, such as `utils::stack` and `base::vector`.

## Value

Returns a `CppMultiset` object referencing a multiset in C++.

## See Also

[cpp\\_set](#), [cpp\\_unordered\\_set](#), [cpp\\_unordered\\_multiset](#).

## Examples

```
s <- cpp_multiset(c(6:9, 6L))
s
# 6 6 7 8 9

insert(s, 4:7)
s
# 4 5 6 6 6 7 7 8 9

print(s, from = 6)
# 6 6 6 7 7 8 9

s <- cpp_multiset(c("world", "hello", "world", "there"))
s
# "hello" "there" "world" "world"

erase(s, "world")
s
# "hello" "there"
```

---

cpp\_priority\_queue      *Create priority queue*

---

## Description

Create a priority queue. Priority queues are hold ordered, non-unique elements.

## Usage

```
cpp_priority_queue(x, sorting = c("descending", "ascending"))
```

## Arguments

x	An integer, numeric, character, or logical vector.
sorting	"descending" (default) arranges elements in descending order with the largest element at the top. "ascending" sorts the elements in the opposite direction, with the smallest element at the top.

## Details

A priority queue is a container, in which the order of the elements depends on their size rather than their time of insertion. As in a stack, elements are removed from the top.

C++ priority queue methods implemented in this package are [emplace](#), [empty](#), [pop](#), [push](#), [size](#), and [top](#). The package also adds various helper functions ([print](#), [sorting](#), [to\\_r](#), [type](#)).

All object-creating methods in this package begin with `cpp_` to avoid clashes with functions from other packages, such as `utils::stack` and `base::vector`.

## Value

Returns a `CppPriorityQueue` object referencing a `priority_queue` in C++.

## See Also

[cpp\\_queue](#), [cpp\\_stack](#).

## Examples

```
q <- cpp_priority_queue(4:6)
q
# First element: 6

emplace(q, 10L)
q
# First element: 10

emplace(q, 3L)
q
# First element: 10
```

```
top(q)
# [1] 10

q <- cpp_priority_queue(4:6, "ascending")
q
# First element: 4

push(q, 10L)
q
# First element: 4
```

---

cpp\_queue

*Create queue*

---

## Description

Create a queue. Queues are first-in, first-out containers.

## Usage

```
cpp_queue(x)
```

## Arguments

x                    An integer, numeric, character, or logical vector.

## Details

The first element added to a queue is the first one to remove.

C++ queue methods implemented in this package are [back](#), [emplace](#), [empty](#), [front](#), [pop](#), [push](#), and [size](#). The package also adds the `==` operator and various helper functions ([print](#), [to\\_r](#), [type](#)).

All object-creating methods in this package begin with `cpp_` to avoid clashes with functions from other packages, such as `utils::stack` and `base::vector`.

## Value

Returns a `CppQueue` object referencing a queue in C++.

## Examples

```
q <- cpp_queue(1:4)
q
# First element: 1

push(q, 9L)
q
# First element: 1
```

```
back(q)
# [1] 9

emplace(q, 10L)
back(q)
# [1] 10
```

---

cpp\_set

*Create set*

---

## Description

Create a set. Sets are containers of unique, sorted elements.

## Usage

```
cpp_set(x)
```

## Arguments

x                    An integer, numeric, character, or logical vector.

## Details

Sets are associative containers. They do not provide random access through an index. I.e., `s[2]` does not return the second element.

C++ set methods implemented in this package are [clear](#), [contains](#), [count](#), [emplace](#), [empty](#), [erase](#), [insert](#), [max\\_size](#), [merge](#), and [size](#). The package also adds the `==` operator and various helper functions ([print](#), [to\\_r](#), [type](#)).

All object-creating methods in this package begin with `cpp_` to avoid clashes with functions from other packages, such as `utils::stack` and `base::vector`.

## Value

Returns a CppSet object referencing a set in C++.

## See Also

[cpp\\_unordered\\_set](#), [cpp\\_multiset](#), [cpp\\_unordered\\_multiset](#).

## Examples

```
s <- cpp_set(6:9)
s
# 6 7 8 9

insert(s, 4:7)
s
# 4 5 6 7 8 9

print(s, from = 6)
# 6 7 8 9

s <- cpp_set(c("world", "hello", "there"))
s
# "hello" "there" "world"

erase(s, "there")
s
# "hello" "world"
```

---

cpp\_stack

*Create stack*

---

## Description

Create a stack. Stacks are last-in, first-out containers.

## Usage

```
cpp_stack(x)
```

## Arguments

x                   An integer, numeric, character, or logical vector.

## Details

The last element added to a stack is the first one to remove.

C++ stack methods implemented in this package are [emplace](#), [empty](#), [pop](#), [push](#), [size](#), and [top](#). The package also adds the `==` operator and various helper functions ([print](#), [to\\_r](#), [type](#)).

All object-creating methods in this package begin with `cpp_` to avoid clashes with functions from other packages, such as `utils::stack` and `base::vector`.

## Value

Returns a `CppStack` object referencing a stack in C++.

**See Also**

[cpp\\_queue](#), [cpp\\_priority\\_queue](#).

**Examples**

```
s <- cpp_stack(4:6)
s
# Top element: 6

emplace(s, 3L)
s
# Top element: 3

push(s, 9L)
s
# Top element: 9

pop(s)
s
# Top element: 3
```

---

cpp\_unordered\_map      *Create unordered map*

---

**Description**

Create an unordered map. Unordered maps are key-value pairs with unique keys.

**Usage**

```
cpp_unordered_map(keys, values)
```

**Arguments**

keys	An integer, numeric, character, or logical vector.
values	An integer, numeric, character, or logical vector.

**Details**

Unordered maps are associative containers. They do not provide random access through an index. I.e. `m[2]` does not return the second element.

Unordered means that the container does not enforce elements to be stored in a particular order. This makes unordered maps in some applications faster than maps.

C++ `unordered_map` methods implemented in this package are [at](#), [bucket\\_count](#), [clear](#), [contains](#), [count](#), [emplace](#), [empty](#), [erase](#), [insert](#), [insert\\_or\\_assign](#), [load\\_factor](#), [max\\_bucket\\_count](#), [max\\_load\\_factor](#), [max\\_size](#), [merge](#), [rehash](#), [reserve](#), [size](#), and [try\\_emplace](#). The package also adds the `==` and `[]` operators and various helper functions ([print](#), [to\\_r](#), [type](#)).

All object-creating methods in this package begin with `cpp_` to avoid clashes with functions from other packages, such as `utils::stack` and `base::vector`.

### Value

Returns a `CppUnorderedMap` object referencing an `unordered_map` in C++.

### See Also

[cpp\\_map](#), [cpp\\_multimap](#), [cpp\\_unordered\\_multimap](#).

### Examples

```
m <- cpp_unordered_map(4:6, seq.int(1, by = 0.5, length.out = 3L))
m
# [6,2] [5,1.5] [4,1]

insert(m, seq.int(100, by = 0.1, length.out = 3L), 14:16)
m
# [16,100.2] [15,100.1] [14,100] [6,2] [5,1.5] [4,1]

print(m, n = 3)
# [16,100.2] [15,100.1] [14,100]

m <- cpp_unordered_map(c("world", "hello", "there"), 4:6)
m
# ["there",6] ["hello",5] ["world",4]

erase(m, "there")
m
# ["hello",5] ["world",4]
```

---

cpp\_unordered\_multimap

*Create unordered multimap*

---

### Description

Create an unordered multimap. Unordered multimaps are key-value pairs with non-unique keys.

### Usage

```
cpp_unordered_multimap(keys, values)
```

### Arguments

<code>keys</code>	An integer, numeric, character, or logical vector.
<code>values</code>	An integer, numeric, character, or logical vector.



## Details

Unordered multimaps are associative containers. They do not provide random access through an index. I.e. `m[2]` does not return the second element.

Unordered means that the container does not enforce elements to be stored in a particular order. This makes unordered multimaps in some applications faster than multimaps.

C++ unordered\_multimap methods implemented in this package are [bucket\\_count](#), [clear](#), [contains](#), [count](#), [emplace](#), [empty](#), [erase](#), [insert](#), [load\\_factor](#), [max\\_bucket\\_count](#), [max\\_load\\_factor](#), [max\\_size](#), [merge](#), [rehash](#), [reserve](#), and [size](#). The package also adds the `==` operator and various helper functions ([print](#), [to\\_r](#), [type](#)).

All object-creating methods in this package begin with `cpp_` to avoid clashes with functions from other packages, such as `utils::stack` and `base::vector`.

## Value

Returns a `CppUnorderedMultimap` object referencing an `unordered_multimap` in C++.

## See Also

[cpp\\_map](#), [cpp\\_unordered\\_map](#), [cpp\\_multimap](#).

## Examples

```
m <- cpp_unordered_multimap(c("world", "hello", "there", "hello"), 4:7)
m
# ["there",6] ["hello",5] ["hello",7] ["world",4]

print(m, n = 2)
#

erase(m, "hello")
m
# ["there",6] ["world",4]

contains(m, "there")
# [1] TRUE
```

---

cpp\_unordered\_multiset

*Create unordered multiset*

---

## Description

Create an unordered multiset. Unordered multisets are containers of non-unique, unsorted elements.

## Usage

```
cpp_unordered_multiset(x)
```

**Arguments**

x                    An integer, numeric, character, or logical vector.

**Details**

Unordered sets are associative containers. They do not provide random access through an index. I.e. `s[2]` does not return the second element.

Unordered means that the container does not enforce elements to be stored in a particular order. This makes unordered multisets in some applications faster than multisets. I.e., elements in unordered multisets are neither unique nor sorted.

C++ `unordered_multiset` methods implemented in this package are [bucket\\_count](#), [clear](#), [contains](#), [count](#), [emplace](#), [empty](#), [erase](#), [insert](#), [load\\_factor](#), [max\\_bucket\\_count](#), [max\\_load\\_factor](#), [max\\_size](#), [merge](#), [rehash](#), [reserve](#), and [size](#). The package also adds the `==` operator and various helper functions ([print](#), [to\\_r](#), [type](#)).

All object-creating methods in this package begin with `cpp_` to avoid clashes with functions from other packages, such as `utils::stack` and `base::vector`.

**Value**

Returns a `CppUnorderedMultiset` object referencing an `unordered_multiset` in C++.

**See Also**

[cpp\\_set](#), [cpp\\_unordered\\_set](#), [cpp\\_multiset](#).

**Examples**

```
s <- cpp_unordered_multiset(c(6:10, 7L))
s
# 10 9 8 7 7 6

insert(s, 4:7)
s
# 5 4 6 6 7 7 7 8 9 10

print(s, n = 3L)
# 5 4 6

erase(s, 6L)
s
# 5 4 7 7 7 8 9 10
```

---

cpp\_unordered\_set      *Create unordered set*

---

### Description

Create an unordered set. Unordered sets are containers of unique, unsorted elements.

### Usage

```
cpp_unordered_set(x)
```

### Arguments

x                      An integer, numeric, character, or logical vector.

### Details

Unordered sets are associative containers. They do not provide random access through an index. I.e. `s[2]` does not return the second element.

Unordered means that the container does not enforce elements to be stored in a particular order. This makes unordered sets in some applications faster than sets. I.e., elements in unordered sets are unique, but not sorted.

C++ `unordered_set` methods implemented in this package are [bucket\\_count](#), [clear](#), [contains](#), [count](#), [emplace](#), [empty](#), [erase](#), [insert](#), [load\\_factor](#), [max\\_bucket\\_count](#), [max\\_load\\_factor](#), [max\\_size](#), [merge](#), [rehash](#), [reserve](#), and [size](#). The package also adds the `==` operator and various helper functions ([print](#), [to\\_r](#), [type](#)).

All object-creating methods in this package begin with `cpp_` to avoid clashes with functions from other packages, such as `utils::stack` and `base::vector`.

### Value

Returns a `CppUnorderedSet` object referencing an `unordered_set` in C++.

### See Also

[cpp\\_set](#), [cpp\\_multiset](#), [cpp\\_unordered\\_multiset](#).

### Examples

```
s <- cpp_unordered_set(6:10)
s
# 10 9 8 7 6

insert(s, 4:7)
s
# 5 4 10 9 8 7 6

print(s, n = 3L)
```

```
# 5 4 10

s <- cpp_unordered_set(c("world", "hello", "there"))
s
# "there" "hello" "world"

erase(s, "there")
s
# "hello" "world"
```

---

cpp\_vector

*Create vector*

---

## Description

Create a vector. Vectors are dynamic, contiguous arrays.

## Usage

```
cpp_vector(x)
```

## Arguments

`x` An integer, numeric, character, or logical vector.

## Details

R vectors are similar to C++ vectors. These sequence containers allow for random access. I.e., you can directly access the fourth element via its index `x[4]`, without iterating through the first three elements before. Vectors are comparatively space-efficient, requiring less RAM per element than many other container types.

One advantage of C++ vectors over R vectors is their ability to reduce the number of copies made during modifications. [reserve](#), e.g., reserves space for future vector extensions.

C++ vector methods implemented in this package are [assign](#), [at](#), [back](#), [capacity](#), [clear](#), [emplace](#), [emplace\\_back](#), [empty](#), [erase](#), [flip](#), [front](#), [insert](#), [max\\_size](#), [pop\\_back](#), [push\\_back](#), [reserve](#), [resize](#), [shrink\\_to\\_fit](#), and [size](#). The package also adds the `==` and `[]` operators and various helper functions ([print](#), [to\\_r](#), [type](#)).

All object-creating methods in this package begin with `cpp_` to avoid clashes with functions from other packages, such as `utils::stack` and `base::vector`.

## Value

Returns a `CppVector` object referencing a vector in C++.

## See Also

[cpp\\_deque](#), [cpp\\_forward\\_list](#), [cpp\\_list](#).

### Examples

```
v <- cpp_vector(4:6)
v
# 4 5 6

push_back(v, 3L)
v
# 4 5 6 3

print(v, from = 3)
# 6 3

print(v, n = -2)
# 3 6

pop_back(v)
v
# 4 5 6
```

---

emplace	<i>Add an element</i>
---------	-----------------------

---

### Description

Add an element to a container by reference in place.

### Usage

```
emplace(x, value, key = NULL, position = NULL)
```

### Arguments

x	A CppSet, CppUnorderedSet, CppMultiset, CppUnorderedMultiset, CppMap, CppUnorderedMap, CppMultimap, CppUnorderedMultimap, CppStack, CppQueue, CppPriorityQueue, CppVector, CppDeque, or CppList object.
value	A value to add to x.
key	A key to add to x. Only relevant for CppMap, CppUnorderedMap, CppMultimap, and CppUnorderedMultimap objects.
position	The index at which to add the element. Only relevant for CppVector, CppDeque, and CppList objects. Indices start at 1.

### Details

Existing container values are not overwritten. I.e. inserting a key-value pair into a map that already contains that key preserves the old value and discards the new one. Use [insert\\_or\\_assign](#) to overwrite values.

emplace and [try\\_emplace](#) produce the same results in the context of this package. [try\\_emplace](#) can be minimally more computationally efficient than `emplace`.

The `emplace` methods only add single elements. Use [insert](#) to add multiple elements in one call.

### Value

Invisibly returns `NULL`.

### See Also

[assign](#), [emplace\\_after](#), [emplace\\_back](#), [emplace\\_front](#), [insert](#), [try\\_emplace](#).

### Examples

```
s <- cpp_set(c(1.5, 2.3, 4.1))
s
# 1.5 2.3 4.1

emplace(s, 3.1)
s
# 1.5 2.3 3.1 4.1

m <- cpp_unordered_map(c("hello", "there"), c(TRUE, FALSE))
m
# ["there",FALSE] ["hello",TRUE]

emplace(m, TRUE, "world")
m
# ["world",TRUE] ["there",FALSE] ["hello",TRUE]

d <- cpp_deque(4:6)
d
# 4 5 6

emplace(d, 9, position = 2)
d
# 4 9 5 6
```

---

emplace\_after

*Add an element*

---

### Description

Add an element to a forward list by reference in place.

### Usage

```
emplace_after(x, value, position)
```

**Arguments**

<code>x</code>	A CppForwardList object.
<code>value</code>	Value to add to <code>x</code> .
<code>position</code>	Index after which to insert the element. Indices start at 1. The function does not perform bounds checks. Indices outside <code>x</code> crash the program.

**Details**

The `emplace` methods only add single elements. Use [insert](#) to add multiple elements in one call.

**Value**

Invisibly returns `NULL`.

**See Also**

[emplace](#), [emplace\\_back](#), [emplace\\_front](#), [insert](#).

**Examples**

```
l <- cpp_forward_list(4:6)
l
# 4 5 6

emplace_after(l, 10L, 2)
l
# 4 5 10 6
```

---

<code>emplace_back</code>	<i>Add an element to the back</i>
---------------------------	-----------------------------------

---

**Description**

Add an element to the back of a container by reference in place.

**Usage**

```
emplace_back(x, value)
```

**Arguments**

<code>x</code>	A CppVector, CppDeque, CppList object.
<code>value</code>	A value to add to <code>x</code> .

**Details**

The `emplace` methods only add single elements. Use [insert](#) to add multiple elements in one call.

**Value**

Invisibly returns NULL.

**See Also**

[emplace](#), [emplace\\_after](#), [emplace\\_front](#), [insert](#), [pop\\_back](#), [push\\_back](#).

**Examples**

```
v <- cpp_vector(4:6)
v
# 4 5 6

emplace_back(v, 12L)
v
# 4 5 6 12
```

---

emplace\_front

*Add an element to the front*

---

**Description**

Add an element to the front of a container by reference in place.

**Usage**

```
emplace_front(x, value)
```

**Arguments**

x	A CppDeque, CppForwardList, or CppList object.
value	A value to add to x.

**Details**

The emplace methods only add single elements. Use [insert](#) to add multiple elements in one call.

**Value**

Invisibly returns NULL.

**See Also**

[emplace](#), [emplace\\_after](#), [emplace\\_back](#), [insert](#), [pop\\_front](#), [push\\_front](#).



**Examples**

```
l <- cpp_forward_list(4:6)
l
# 4 5 6

emplace_front(l, 12L)
l
# 12 4 5 6
```

---

empty

*Check emptiness*

---

**Description**

Check, if a container is empty, i.e. does not contain elements.

**Usage**

```
empty(x)
```

**Arguments**

x                    A cppcontainers object.

**Value**

Returns TRUE, if the container is empty, and FALSE otherwise.

**Examples**

```
v <- cpp_vector(4:6)
v
# 4 5 6

clear(v)
empty(v)
# [1] TRUE
```

---

erase *Erase elements*

---

### Description

Delete elements from a container by reference.

### Usage

```
erase(x, values = NULL, from = NULL, to = NULL)
```

### Arguments

x	A CppSet, CppUnorderedSet, CppMultiset, CppUnorderedMultiset, CppMap, CppUnorderedMap, CppMultimap, CppUnorderedMultimap, CppVector, CppDeque, or CppList object.
values	A vector of values to delete from x in CppSet, CppUnorderedSet, CppMultiset, and CppUnorderedMultiset objects and keys in CppMap, CppUnorderedMap, CppMultimap, and CppUnorderedMultimap objects. Ignored for other classes.
from	Index of the first element to be deleted in CppVector, CppDeque, and CppList objects. Ignored for other classes.
to	Index of the last element to be deleted in CppVector, CppDeque, and CppList objects. Ignored for other classes.

### Value

Invisibly returns NULL.

### See Also

[clear](#), [empty](#), [erase\\_after](#), [remove..](#)

### Examples

```
s <- cpp_multiset(c(2, 2.1, 3, 3, 4.3, 6))
s
# 2 2.1 3 3 4.3 6

erase(s, c(2, 3))
s
# 2.1 4.3 6

m <- cpp_unordered_multimap(c(2:3, 3L), c("hello", "there", "world"))
m
# [3,"world"] [3,"there"] [2,"hello"]

erase(m, 2L)
m
```

```
# [3,"world"] [3,"there"]

d <- cpp_deque(4:9)
d
# 4 5 6 7 8 9

erase(d, from = 2, to = 3)
d
# 4 7 8 9
```

---

erase\_after

*Erase elements*

---

### Description

Delete elements from a forward list by reference.

### Usage

```
erase_after(x, from, to)
```

### Arguments

x	A CppForwardList object.
from	Index after which to delete.
to	Index until including which to delete. Indices start at 1. The function does not perform bounds checks. Indices outside x crash the program.

### Value

Invisibly returns NULL.

### See Also

[clear](#), [empty](#), [erase](#), [remove](#)..

### Examples

```
l <- cpp_forward_list(4:9)
l
# 4 5 6 7 8 9

erase_after(l, 2L, 4L)
l
# 4 5 8 9
```

---

flip	<i>Toggle boolean values</i>
------	------------------------------

---

**Description**

Toggle boolean values in a vector.

**Usage**

```
flip(x)
```

**Arguments**

x                    A CppVector object of type boolean.

**Details**

Sets TRUE to FALSE and FALSE to TRUE.

**Value**

Invisibly returns NULL.

**See Also**

[cpp\\_vector](#).

**Examples**

```
v <- cpp_vector(c(TRUE, TRUE, FALSE))
v
# TRUE TRUE FALSE

flip(v)
v
# FALSE FALSE TRUE
```

---

front	<i>Access first element</i>
-------	-----------------------------

---

**Description**

Access first element in a container without removing it.

**Usage**

```
front(x)
```

**Arguments**

x                   A CppQueue, CppVector, CppDeque, CppForwardList, or CppList object.

**Value**

Returns the front element.

**See Also**

[back](#), [emplace\\_front](#), [pop](#), [pop\\_front](#), [push\\_front](#).

**Examples**

```
q <- cpp_queue(4:6)
q
# First element: 4

front(q)
# [1] 4

q
# First element: 4

v <- cpp_vector(c(TRUE, FALSE, FALSE))
v
# TRUE FALSE FALSE

front(v)
# [1] TRUE
```

---

insert	<i>Add elements</i>
--------	---------------------

---

**Description**

Add elements to a container by reference.

**Usage**

```
insert(x, values, keys = NULL, position = NULL)
```

**Arguments**

x	A CppSet, CppUnorderedSet, CppMultiset, CppUnorderedMultiset, CppMap, CppUnorderedMap, CppMultimap, CppUnorderedMultimap, CppVector, CppDeque, or CppList object.
values	Values to add to x.
keys	Keys to add to x. Only relevant for CppMap, CppUnorderedMap, CppMultimap, and CppUnorderedMultimap objects.
position	Index at which to insert elements into x. Only relevant for CppVector, CppDeque, and CppList objects. Indices start at 1.

**Details**

Existing container values are not overwritten. I.e. inserting a key-value pair into a map that already contains that key preserves the old value and discards the new one. Use [insert\\_or\\_assign](#) to overwrite values.

**Value**

Invisibly returns NULL.

**See Also**

[assign](#), [emplace](#), [insert\\_after](#), [insert\\_or\\_assign](#), [push](#).

**Examples**

```
s <- cpp_multiset(4:6)
s
# 4 5 6

insert(s, 6:7)
s
# 4 5 6 6 7

m <- cpp_map(c("hello", "there", "world"), 9:11)
m
```

```
# ["hello",9] ["there",10] ["world",11]

insert(m, 12L, "there")
m
# ["hello",9] ["there",10] ["world",11]
```

---

insert\_after

*Add elements*

---

## Description

Add elements to a forward list by reference.

## Usage

```
insert_after(x, values, position = NULL)
```

## Arguments

x	A CppForwardList object.
values	Values to add to x.
position	Index behind which to insert elements.

## Value

Invisibly returns NULL.

## See Also

[emplace](#), [emplace\\_after](#), [insert](#), [insert\\_or\\_assign](#).

## Examples

```
v <- cpp_forward_list(4:6)
v
# 4 5 6

insert_after(v, 10:11, 2)
v
# 4 5 10 11 6
```

---

insert\_or\_assign      *Add or overwrite elements*

---

### Description

Add elements to a container by reference. Overwrites existing container values tied to the same keys.

### Usage

```
insert_or_assign(x, values, keys)
```

### Arguments

x	A CppMap or CppUnorderedMap object.
values	Values to add to x.
keys	Keys to add to x.

### Details

Use [insert](#) to avoid overwriting values.

### Value

Invisibly returns NULL.

### See Also

[insert](#), [insert\\_after](#), [emplace](#), [try\\_emplace](#).

### Examples

```
m <- cpp_map(4:6, 9:11)
m
# [4,9] [5,10] [6,11]

insert_or_assign(m, 12:13, 6:7)
m
# [4,9] [5,10] [6,12] [7,13]
```



---

load_factor	<i>Get the mean number of elements per bucket</i>
-------------	---

---

**Description**

Get the mean number of elements per bucket.

**Usage**

```
load_factor(x)
```

**Arguments**

x                    A CppUnorderedSet, CppUnorderedMultiset, CppUnorderedMap, or CppUnordered-Multimap object.

**Value**

Returns a numeric.

**See Also**

[bucket\\_count](#), [max\\_bucket\\_count](#), [max\\_load\\_factor](#).

**Examples**

```
s <- cpp_unordered_set(6:9)
load_factor(s)
# [1] 0.3076923
```

---

max_bucket_count	<i>Get the maximum number of buckets</i>
------------------	--

---

**Description**

Obtain the maximum number of buckets the container can hold.

**Usage**

```
max_bucket_count(x)
```

**Arguments**

x                    A CppUnorderedSet, CppUnorderedMultiset, CppUnorderedMap, or CppUnordered-Multimap object.

**Value**

Returns a numeric.

**See Also**

[bucket\\_count](#), [load\\_factor](#), [max\\_load\\_factor](#).

**Examples**

```
s <- cpp_unordered_set(6:10)
max_bucket_count(s)
# [1] 1.152922e+18
```

---

max\_load\_factor

*Get or set the maximum load factor*

---

**Description**

Get or set the maximum load factor by reference, i.e. the number of elements per bucket.

**Usage**

```
max_load_factor(x, max_load = NULL)
```

**Arguments**

x	A CppUnorderedSet, CppUnorderedMultiset, CppUnorderedMap, or CppUnordered-Multimap object.
max_load	The containers maximum load factor. If NULL, the function returns the container's current maximum load factor. Passing a number sets the maximum load factor to that value.

**Value**

Returns a numeric, if max\_load is NULL. Invisibly returns NULL, if max\_load is numeric.

**See Also**

[bucket\\_count](#), [load\\_factor](#), [max\\_bucket\\_count](#).

**Examples**

```
s <- cpp_unordered_set(4:6)
max_load_factor(s)
# [1] 1

max_load_factor(s, 3)
max_load_factor(s)
# [1] 3
```

---

max\_size

*Get maximum container size*

---

**Description**

Obtain the maximum number of elements the container can hold.

**Usage**

```
max_size(x)
```

**Arguments**

x            A CppSet, CppUnorderedSet, CppMultiset, CppUnorderedMultiset, CppMap, CppUnorderedMap, CppMultimap, CppUnorderedMultimap, CppVector, CppDeque, CppForwardList, or CppList object.

**Value**

Returns a numeric.

**See Also**

[capacity](#), [max\\_bucket\\_count](#), [max\\_load\\_factor](#), [size](#).

**Examples**

```
s <- cpp_deque(4:6)
s
# 4 5 6

max_size(s)
# [1] 4.611686e+18
```

---

merge	<i>Merge two objects</i>
-------	--------------------------

---

### Description

Merge two objects by reference.

### Usage

```
merge(x, y, ...)
```

### Arguments

x	A CppSet, CppUnorderedSet, CppMultiset, CppUnorderedMultiset, CppMap, CppUnorderedMap, CppMultimap, CppUnorderedMultimap, CppForwardList, or CppList object.
y	A CppSet, CppUnorderedSet, CppMultiset, CppUnorderedMultiset, CppMap, CppUnorderedMap, CppMultimap, CppUnorderedMultimap, CppForwardList, or CppList object of the same class and data type as x.
...	Ignored. Only included for compatibility with generic base::merge method.

### Details

In containers enforcing uniqueness (CppSet, CppUnorderedSet, CppMap, CppUnorderedMap), the function merges elements from y that are not in x into x and deletes them from y. In other container types, it transfers all elements.

### Value

Invisibly returns NULL.

### See Also

[assign](#), [emplace](#), [insert](#).

### Examples

```
x <- cpp_set(c("hello", "there"))
y <- cpp_set(c("hello", "world"))

merge(x, y)
x
# "hello" "there" "world"
y
# "hello"

x <- cpp_forward_list(c(1, 3, 4, 3))
y <- cpp_forward_list(c(2, 3, 5))
```

```
merge(x, y)
x
# 1 2 3 3 4 3 5
y
#
```

---

pop

*Remove top element*

---

### Description

Remove top element in a stack or priority queue or the first element in a queue by reference.

### Usage

```
pop(x)
```

### Arguments

x                   A CppStack, CppQueue, or CppPriorityQueue object.

### Details

In a stack, it is the last inserted element. In a queue, it is the first inserted element. In a descending (ascending) priority queue, it is the largest (smallest) value.

### Value

Invisibly returns NULL.

### See Also

[back](#), [emplace](#), [front](#), [push](#), [top](#).

### Examples

```
s <- cpp_stack(4:6)
s
# Top element: 6

pop(s)
s
# Top element: 5

q <- cpp_queue(4:6)
q
# First element: 4
```

```
pop(q)
q
# First element: 5

p <- cpp_priority_queue(4:6)
p
# First element: 6

pop(p)
p
# First element: 5
```

---

pop_back	<i>Remove an element from the back</i>
----------	--

---

### Description

Remove an element from the back of the container by reference.

### Usage

```
pop_back(x)
```

### Arguments

x                    A CppVector, CppDeque, or CppList object.

### Value

Invisibly returns NULL.

### See Also

[back](#), [emplace\\_back](#), [pop](#), [pop\\_front](#), [push\\_back](#).

### Examples

```
l <- cpp_list(4:6)
l
# 4 5 6

pop_back(l)
l
# 4 5
```

---

pop_front	<i>Remove an element from the front</i>
-----------	---

---

**Description**

Remove an element from the front of the container by reference.

**Usage**

```
pop_front(x)
```

**Arguments**

x                    A CppDeque, CppForwardList, or CppList object.

**Value**

Invisibly returns NULL.

**See Also**

[emplace\\_front](#), [front](#), [pop](#), [pop\\_back](#), [push\\_front](#).

**Examples**

```
d <- cpp_deque(4:6)
d
# 4 5 6

pop_front(d)
d
# 5 6
```

---

print	<i>Print container data</i>
-------	-----------------------------

---

**Description**

Print the data in a container.

**Usage**

```
print(x, ...)
```

## Arguments

<code>x</code>	A <code>cppcontainers</code> object.
<code>...</code>	An ellipsis for compatibility with the generic method. Accepts the parameters <code>n</code> , <code>from</code> , and <code>to</code> . See <a href="#">to_r</a> for their effects. A difference to <a href="#">to_r</a> is that their omission does not induce the function to print all elements, but to print the first 100 elements. Stacks, queues, and priority queues ignore the ellipsis and only print the top or first element.

## Details

`print` has no side effects. Unlike [to\\_r](#), it does not remove elements from stacks or queues.

## Value

Invisibly returns `NULL`.

## See Also

[sorting](#), [to\\_r](#), [type](#).

## Examples

```
s <- cpp_set(4:9)

print(s)
# 4 5 6 7 8 9

print(s, n = 3)
# 4 5 6

print(s, n = -3)
# 9 8 7

print(s, from = 5, to = 7)
# 5 6 7

v <- cpp_vector(4:9)

print(v, n = 2)
# 4 5

print(v, from = 2, to = 3)
# 5 6

print(v, from = 3)
# 6 7 8 9
```



---

push	<i>Add elements</i>
------	---------------------

---

### Description

Add elements to the top of a stack, to the back of a queue, or to a priority queue by reference.

### Usage

```
push(x, values)
```

### Arguments

x	A CppStack, CppQueue, or CppPriorityQueue object.
values	Values to add to x.

### Details

The method iterates through values starting at the front of the vector. I.e., the last element of values is added last.

### Value

Invisibly returns NULL.

### See Also

[back](#), [emplace](#), [front](#), [pop](#), [push](#), [top](#).

### Examples

```
s <- cpp_stack(1:4)
s
# Top element: 4

push(s, 8:9)
s
# Top element: 9
```

---

`push_back`*Add an element to the back*

---

**Description**

Add an element to the back of a container by reference.

**Usage**

```
push_back(x, value)
```

**Arguments**

<code>x</code>	A CppVector, CppDeque, or CppList object.
<code>value</code>	A value to add to <code>x</code> .

**Value**

Invisibly returns NULL.

**See Also**

[back](#), [emplace\\_back](#), [insert](#), [pop\\_back](#), [push\\_front](#).

**Examples**

```
v <- cpp_vector(4:6)
v
# 4 5 6

push_back(v, 14L)
v
# 4 5 6 14
```

---

`push_front`*Add an element to the front*

---

**Description**

Add an element to the front of a container by reference.

**Usage**

```
push_front(x, value)
```

**Arguments**

x                    A CppDeque, CppForwardList, or CppList object.  
 value                A value to add to x.

**Value**

Invisibly returns NULL.

**See Also**

[emplace\\_front](#), [front](#), [insert](#), [pop\\_front](#), [push\\_back](#).

**Examples**

```
d <- cpp_deque(4:6)
d
# 4 5 6

push_front(d, 14L)
d
# 14 4 5 6
```

---

 rehash

*Set minimum bucket count and rehash*


---

**Description**

Set a container's minimum bucket count and rehash by reference.

**Usage**

```
rehash(x, n = 0)
```

**Arguments**

x                    A CppUnorderedSet, CppUnorderedMultiset, CppUnorderedMap, or CppUnordered-Multimap object.  
 n                    The minimum number of buckets. A value of 0 forces an unconditional rehash.

**Value**

Invisibly returns NULL.

**See Also**

[bucket\\_count](#), [load\\_factor](#), [max\\_bucket\\_count](#), [max\\_load\\_factor](#), [reserve](#).

**Examples**

```
s <- cpp_unordered_set(4:6)
rehash(s)
rehash(s, 3)
```

---

remove.

*Remove elements*

---

**Description**

Remove elements from a container by reference.

**Usage**

```
remove.(x, value)
```

**Arguments**

x	A CppForwardList or CppList object.
value	A value to delete from x.

**Details**

The method ends with a dot to avoid compatibility issues with the generic base::remove.

**Value**

Invisibly returns NULL.

**See Also**

[clear](#), [empty](#), [erase](#).

**Examples**

```
l <- cpp_forward_list(4:6)
l
# 4 5 6

remove.(l, 5L)
l
# 4 6
```

---

reserve	<i>Reserve space</i>
---------	----------------------

---

### Description

Reserve space for the container by reference.

### Usage

```
reserve(x, n)
```

### Arguments

x	A CppUnorderedSet, CppUnorderedMultiset, CppUnorderedMap, CppUnorderedMultimap, or CppVector object.
n	The minimum number of elements per bucket.

### Details

In case of a CppUnorderedSet, CppUnorderedMultiset, CppUnorderedMap, CppUnorderedMultimap, the method sets the number of buckets to be able to hold at least n elements and rehashes. In case of a CppVector, the method sets the capacity to n.

### Value

Invisibly returns NULL.

### See Also

[bucket\\_count](#), [capacity](#), [load\\_factor](#), [max\\_bucket\\_count](#), [max\\_load\\_factor](#).

### Examples

```
s <- cpp_unordered_set(4:6)
bucket_count(s)
# [1] 13
reserve(s, 3)
bucket_count(s)
# [1] 5

v <- cpp_vector(4:6)
capacity(v)
# [1] 3
reserve(v, 10)
capacity(v)
# [1] 10
```

---

`resize`*Alter the container size*

---

**Description**

Alter the size of the container by reference.

**Usage**

```
resize(x, size, value = NULL)
```

**Arguments**

<code>x</code>	A <code>cppcontainers</code> object.
<code>size</code>	The new size of the container.
<code>value</code>	The value of new elements. It defaults to <code>0</code> for integers and doubles, to <code>""</code> for strings, and to <code>FALSE</code> for booleans.

**Details**

If the new size is larger than the former size, the function sets newly added elements in the back to `value`.

**Value**

Invisibly returns `NULL`.

**Examples**

```
v <- cpp_vector(4:9)
v
# 4 5 6 7 8 9

size(v)
# 6

resize(v, 10)
v
# 4 5 6 7 8 9 0 0 0 0

resize(v, 3)
v
# 4 5 6
```

---

reverse	<i>Reverse element order</i>
---------	------------------------------

---

**Description**

Reverses the order of the elements in the container by reference.

**Usage**

```
reverse(x)
```

**Arguments**

x                    A CppForwardList or CppList object.

**Value**

Invisibly returns NULL.

**Examples**

```
l <- cpp_forward_list(4:9)
l
# 4 5 6 7 8 9

reverse(l)
l
#
```

---

shrink_to_fit	<i>Shrink container capacity to size</i>
---------------	--

---

**Description**

Shrink the capacity of a container to its size by reference.

**Usage**

```
shrink_to_fit(x)
```

**Arguments**

x                    A CppVector or CppDeque object.

**Details**

The capacity is the space, in terms of the number of elements, reserved for a container. The size is the number of elements in the container.

**Value**

Invisibly returns NULL.

**See Also**

[capacity](#), [reserve](#), [size](#).

**Examples**

```
v <- cpp_vector(4:6)
capacity(v)
# [1] 3

reserve(v, 10)
capacity(v)
# [1] 10

shrink_to_fit(v)
capacity(v)
# [1] 3
```

---

size

*Get container size*

---

**Description**

Obtain the number of elements in a container.

**Usage**

```
size(x)
```

**Arguments**

x            A CppSet, CppUnorderedSet, CppMultiset, CppUnorderedMultiset, CppMap, CppUnorderedMap, CppMultimap, CppUnorderedMultimap, CppStack, CppQueue, CppPriorityQueue, CppVector, CppDeque, or CppList object.

**Value**

Returns a numeric.



**See Also**

[bucket\\_count](#), [capacity](#), [load\\_factor](#), [max\\_size](#), [resize](#).

**Examples**

```
s <- cpp_unordered_set(4:6)
s
# 6 5 4

size(s)
# [1] 3
```

---

sort

*Sort elements*

---

**Description**

Sorts the elements in a container by reference.

**Usage**

```
sort(x, decreasing, ...)
```

**Arguments**

x	A CppForwardList or CppList object.
decreasing	Ignored.
...	Ignored.

**Details**

decreasing and ... are only included for compatibility with the generic base::sort method and have no effect.

**Value**

Invisibly returns NULL.

**See Also**

[sorting](#), [unique](#).

**Examples**

```
l <- cpp_forward_list(c(3, 2, 4))
l
# 3 2 4

sort(l)
l
# 2 3 4
```

---

sorting

*Print the sorting order*

---

**Description**

Print the sorting order of a priority queue.

**Usage**

```
sorting(x)
```

**Arguments**

x                    An CppPriorityQueue object.

**Value**

Returns "ascending" or "descending".

**See Also**

[cpp\\_priority\\_queue](#), [sort](#).

**Examples**

```
q <- cpp_priority_queue(4:6)
sorting(q)
# [1] "descending"

q <- cpp_priority_queue(4:6, "ascending")
sorting(q)
# [1] "ascending"
```

---

splice	<i>Move elements</i>
--------	----------------------

---

**Description**

Move elements from one list to another list by reference.

**Usage**

```
splice(x, y, x_position, y_from, y_to)
```

**Arguments**

x	A CppList object to which to add elements.
y	A CppList object, of the same data type as x, from which to extract elements.
x_position	Index at which to insert elements in x.
y_from	Index of the first element to extract from y.
y_to	Index of the last element to extract from y.

**Value**

Invisibly returns NULL.

**See Also**

[merge](#), [splice\\_after](#).

**Examples**

```
x <- cpp_list(4:9)
x
# 4 5 6 7 8 9

y <- cpp_list(10:12)
y
# 10 11 12

splice(x, y, 3, 2, 3)
x
# 4 5 11 12 6 7 8 9
y
# 10
```

---

`splice_after`*Move elements*

---

**Description**

Move elements from one forward list to another forward list by reference.

**Usage**

```
splice_after(x, y, x_position, y_from, y_to)
```

**Arguments**

<code>x</code>	A CppForwardList object to which to add elements.
<code>y</code>	A CppForwardList object, of the same data type as <code>x</code> , from which to extract elements.
<code>x_position</code>	Index after which to insert elements in <code>x</code> .
<code>y_from</code>	Index after which to extract elements from <code>y</code> .
<code>y_to</code>	Index of the last element to extract from <code>y</code> .

**Details**

Indices start at 1, which is also the minimum value permitted. Thus, the current implementation in this package does not allow to move the first element of `y`.

**Value**

Invisibly returns `NULL`.

**See Also**

[merge](#), [splice](#).

**Examples**

```
x <- cpp_forward_list(4:9)
x
# 4 5 6 7 8 9

y <- cpp_forward_list(10:12)
y
# 10 11 12

splice_after(x, y, 3, 1, 3)
x
# 4 5 6 11 12 7 8 9
y
# 10
```

---

top	<i>Access top element</i>
-----	---------------------------

---

**Description**

Access the top element in a container without removing it.

**Usage**

```
top(x)
```

**Arguments**

x                    A CppStack or CppPriorityQueue object.

**Value**

Returns the top element.

**See Also**

[emplace](#), [pop](#), [push](#).

**Examples**

```
s <- cpp_stack(1:4)
s
# Top element: 4

top(s)
# [1] 4

s
# Top element: 4
```

---

to_r	<i>Export data to R</i>
------	-------------------------

---

**Description**

Export C++ data to an R object.

**Usage**

```
to_r(x, n = NULL, from = NULL, to = NULL)
```

## Arguments

x	A cppcontainers object.
n	The number of elements to export. If n is positive it exports elements starting at the front of the container. If n is negative, it starts at the back. Negative values only work on CppSet, CppMultiset, CppMap, CppMultimap, CppVector, CppDeque, and CppList objects.
from	The first value in CppSet, CppMultiset, CppMap, CppMultimap objects to export. If it is not a member of x, the export starts at the subsequent value. In a CppVector or CppDeque object, from marks the index of the first element to export. Ignored for other classes.
to	The last value in CppSet, CppMultiset, CppMap, CppMultimap objects to export. If it is not a member of x, the export ends with the prior value. In a CppVector or CppDeque object, from marks the index of the last element to export. Ignored for other classes.

## Details

to\_r has side effects, when applied to stacks, queues, or priority queues. These container types are not iterable. Hence, to\_r **removes elements from the CppStack, CppQueue, and CppPriorityQueue objects when exporting them to R**. When n is specified, the method removes the top n elements from a stack or priority queue or the first n elements from a queue. Otherwise, it removes all elements. Other container types, like sets, etc., are unaffected.

## Value

Returns a vector in case of CppSet, CppUnorderedSet, CppMultiset, CppUnorderedMultiset, CppStack, CppQueue, CppPriorityQueue, CppVector, CppDeque, CppForwardList, and CppList objects. Returns a data frame in case of CppMap, CppUnorderedMap, CppMultimap, and CppUnorderedMultimap objects.

## See Also

[print](#), [sorting](#), [type](#).

## Examples

```
s <- cpp_set(11:20)
to_r(s)
# [1] 11 12 13 14 15 16 17 18 19 20

to_r(s, n = 4)
# [1] 11 12 13 14

to_r(s, n = -4)
# [1] 20 19 18 17

to_r(s, from = 14)
# [1] 14 15 16 17 18 19 20
```

```
to_r(s, to = 18)
# [1] 11 12 13 14 15 16 17 18

to_r(s, from = 14, to = 18)
# [1] 14 15 16 17 18

m <- cpp_unordered_multimap(c("hello", "hello", "there"), 4:6)
to_r(m)
#   key value
# 1 there    6
# 2 hello    4
# 3 hello    5

s <- cpp_stack(11:20)
to_r(s, n = 3)
# [1] 20 19 18
s
# Top element: 17
```

---

try\_emplace

*Add an element*

---

## Description

Add an element to a container by reference in place, if it does not exist yet.

## Usage

```
try_emplace(x, value, key)
```

## Arguments

x	A CppMap or CppUnorderedMap object.
value	A value to add to x.
key	A key to add to x.

## Details

Existing container values are not overwritten. I.e., inserting a key-value pair into a map that already contains that key preserves the old value and discards the new one. Use [insert\\_or\\_assign](#) to overwrite values.

[emplace](#) and `try_emplace` produce the same results in the context of this package. `try_emplace` can be minimally more computationally efficient than [emplace](#).

## Value

Invisibly returns NULL.

**See Also**

[emplace](#), [emplace\\_after](#), [emplace\\_back](#), [emplace\\_front](#), [insert](#), [insert\\_or\\_assign](#).

**Examples**

```
m <- cpp_map(4:6, 9:11)
m
# [4,9] [5,10] [6,11]

try_emplace(m, 13L, 8L)
m
# [4,9] [5,10] [6,11] [8,13]

try_emplace(m, 12L, 4L)
m
# [4,9] [5,10] [6,11] [8,13]
```

---

type

*Get data type*

---

**Description**

Obtain the data type of a container.

**Usage**

```
type(x)
```

**Arguments**

x                    A cppcontainers object.

**Details**

The available types are integer, double, string, and boolean. They correspond to the integer, numeric/ double, character, and logical types in R.

**Value**

A named character vector for CppMap, CppUnorderedMap, CppMultimap, and CppUnorderedMultimap objects. A character otherwise.

**See Also**

[print](#), [sorting](#), [to\\_r](#).



## Examples

```
s <- cpp_set(4:6)
type(s)
# [1] "integer"

m <- cpp_unordered_map(c("hello", "world"), c(0.5, 1.5))
type(m)
#      key    value
# "string" "double"
```

---

unique

*Delete consecutive duplicates*

---

## Description

Erases consecutive duplicated values from the container by reference.

## Usage

```
unique(x, incomparables, ...)
```

## Arguments

x	A CppForwardList or CppList object.
incomparables	Ignored.
...	Ignored.

## Details

Duplicated, non-consecutive elements are not removed.

incomparables and ... are only included for compatibility with the generic base::unique method and have no effect.

## Value

Returns the number of deleted elements.

## See Also

[erase](#), [remove](#)., [sort](#).

**Examples**

```
l <- cpp_forward_list(c(4, 5, 6, 6, 4))
l
# 4 5 6 6 4

unique(l)
# [1] 1
l
# 4 5 6 4
```

---

[,CppMap-method      *Access or insert elements without bounds checking*

---

**Description**

Read or insert a value by key in a CppMap or CppUnorderedMap. Read a value by index in a CppVector or CppDeque.

**Usage**

```
## S4 method for signature 'CppMap'
x[i]

## S4 method for signature 'CppUnorderedMap'
x[i]

## S4 method for signature 'CppVector'
x[i]

## S4 method for signature 'CppDeque'
x[i]
```

**Arguments**

*x*                    A CppMap, CppUnorderedMap, CppVector, or CppDeque object.  
*i*                     A key (CppMap, CppUnorderedMap) or index (CppVector, CppDeque).

**Details**

In the two associative container types (CppMap, CppUnorderedMap), [] accesses a value by its key. If the key does not exist, it enters the key with a default value into the container. The default value is 0 for integer and double, an empty string for string, and FALSE for boolean.

In the two sequence container types (CppVector, CppDeque), [] accesses a value by its index. If the index is outside the container, this crashes the program.

[at](#) and [] both access elements. Unlike [], [at](#) checks the bounds of the container and throws an error, if the element does not exist.

**Value**

Returns the value associated with *i*.

**See Also**

[at](#), [back](#), [contains](#), [front](#), [top](#).

**Examples**

```
m <- cpp_map(4:6, seq.int(0, 1, by = 0.5))
m
# [4,0] [5,0.5] [6,1]

m[6L]
# [1] 1

m
# [4,0] [5,0.5] [6,1]

m[8L]
# [1] 0

m
# [4,0] [5,0.5] [6,1] [8,0]

v <- cpp_vector(4:6)
v
# 4 5 6

v[1L]
# [1] 4

v
# 4 5 6
```

# Index

[==](#), [12–15](#), [17](#), [18](#), [20–23](#), [25–28](#)  
[==](#), [CppDeque](#), [CppDeque-method](#)  
    ([==](#), [CppSet](#), [CppSet-method](#)), [3](#)  
[==](#), [CppForwardList](#), [CppForwardList-method](#)  
    ([==](#), [CppSet](#), [CppSet-method](#)), [3](#)  
[==](#), [CppList](#), [CppList-method](#)  
    ([==](#), [CppSet](#), [CppSet-method](#)), [3](#)  
[==](#), [CppMap](#), [CppMap-method](#)  
    ([==](#), [CppSet](#), [CppSet-method](#)), [3](#)  
[==](#), [CppMultimap](#), [CppMultimap-method](#)  
    ([==](#), [CppSet](#), [CppSet-method](#)), [3](#)  
[==](#), [CppMultiset](#), [CppMultiset-method](#)  
    ([==](#), [CppSet](#), [CppSet-method](#)), [3](#)  
[==](#), [CppQueue](#), [CppQueue-method](#)  
    ([==](#), [CppSet](#), [CppSet-method](#)), [3](#)  
[==](#), [CppSet](#), [CppSet-method](#), [3](#)  
[==](#), [CppStack](#), [CppStack-method](#)  
    ([==](#), [CppSet](#), [CppSet-method](#)), [3](#)  
[==](#), [CppUnorderedMap](#), [CppUnorderedMap-method](#)  
    ([==](#), [CppSet](#), [CppSet-method](#)), [3](#)  
[==](#), [CppUnorderedMultimap](#), [CppUnorderedMultimap-method](#)  
    ([==](#), [CppSet](#), [CppSet-method](#)), [3](#)  
[==](#), [CppUnorderedMultiset](#), [CppUnorderedMultiset-method](#)  
    ([==](#), [CppSet](#), [CppSet-method](#)), [3](#)  
[==](#), [CppUnorderedSet](#), [CppUnorderedSet-method](#)  
    ([==](#), [CppSet](#), [CppSet-method](#)), [3](#)  
[==](#), [CppVector](#), [CppVector-method](#)  
    ([==](#), [CppSet](#), [CppSet-method](#)), [3](#)  
[\[](#), [6](#), [11](#), [12](#), [15](#), [23](#), [28](#)  
[\[](#), [CppDeque-method](#) ([\[](#), [CppMap-method](#)), [66](#)  
[\[](#), [CppMap-method](#), [66](#)  
[\[](#), [CppUnorderedMap-method](#)  
    ([\[](#), [CppMap-method](#)), [66](#)  
[\[](#), [CppVector-method](#) ([\[](#), [CppMap-method](#)), [66](#)  
[assign](#), [5](#), [12–14](#), [28](#), [30](#), [38](#), [44](#)  
[assign](#), [CppDeque-method](#) ([assign](#)), [5](#)  
[assign](#), [CppForwardList-method](#) ([assign](#)), [5](#)  
[assign](#), [CppList-method](#) ([assign](#)), [5](#)  
[assign](#), [CppVector-method](#) ([assign](#)), [5](#)  
[at](#), [6](#), [6](#), [11](#), [12](#), [15](#), [23](#), [28](#), [66](#), [67](#)  
[at](#), [CppDeque-method](#) ([at](#)), [6](#)  
[at](#), [CppMap-method](#) ([at](#)), [6](#)  
[at](#), [CppUnorderedMap-method](#) ([at](#)), [6](#)  
[at](#), [CppVector-method](#) ([at](#)), [6](#)  
[back](#), [6](#), [7](#), [11](#), [12](#), [14](#), [20](#), [28](#), [37](#), [45](#), [46](#), [49](#),  
    [50](#), [67](#)  
[back](#), [CppDeque-method](#) ([back](#)), [7](#)  
[back](#), [CppList-method](#) ([back](#)), [7](#)  
[back](#), [CppQueue-method](#) ([back](#)), [7](#)  
[back](#), [CppVector-method](#) ([back](#)), [7](#)  
[bucket\\_count](#), [8](#), [23](#), [25–27](#), [41](#), [42](#), [51](#), [53](#), [57](#)  
[bucket\\_count](#), [CppUnorderedMap-method](#)  
    ([bucket\\_count](#)), [8](#)  
[bucket\\_count](#), [CppUnorderedMultimap-method](#)  
    ([bucket\\_count](#)), [8](#)  
[bucket\\_count](#), [CppUnorderedMultiset-method](#)  
    ([bucket\\_count](#)), [8](#)  
[bucket\\_count](#), [CppUnorderedSet-method](#)  
    ([bucket\\_count](#)), [8](#)  
[capacity](#), [9](#), [28](#), [43](#), [53](#), [56](#), [57](#)  
[capacity](#), [CppVector-method](#) ([capacity](#)), [9](#)  
[clear](#), [10](#), [12–15](#), [17](#), [18](#), [21](#), [23](#), [25–28](#), [34](#),  
    [35](#), [52](#)  
[clear](#), [CppDeque-method](#) ([clear](#)), [10](#)  
[clear](#), [CppForwardList-method](#) ([clear](#)), [10](#)  
[clear](#), [CppList-method](#) ([clear](#)), [10](#)  
[clear](#), [CppMap-method](#) ([clear](#)), [10](#)  
[clear](#), [CppMultimap-method](#) ([clear](#)), [10](#)  
[clear](#), [CppMultiset-method](#) ([clear](#)), [10](#)  
[clear](#), [CppSet-method](#) ([clear](#)), [10](#)  
[clear](#), [CppUnorderedMap-method](#) ([clear](#)), [10](#)  
[clear](#), [CppUnorderedMultimap-method](#)  
    ([clear](#)), [10](#)  
[clear](#), [CppUnorderedMultiset-method](#)  
    ([clear](#)), [10](#)  
[clear](#), [CppUnorderedSet-method](#) ([clear](#)), [10](#)  
[clear](#), [CppVector-method](#) ([clear](#)), [10](#)

- contains, [5](#), [6](#), [10](#), [12](#), [15](#), [17](#), [18](#), [21](#), [23](#), [25–27](#), [67](#)
- contains, CppMap-method (contains), [10](#)
- contains, CppMultimap-method (contains), [10](#)
- contains, CppMultiset-method (contains), [10](#)
- contains, CppSet-method (contains), [10](#)
- contains, CppUnorderedMap-method (contains), [10](#)
- contains, CppUnorderedMultimap-method (contains), [10](#)
- contains, CppUnorderedMultiset-method (contains), [10](#)
- contains, CppUnorderedSet-method (contains), [10](#)
- count, [11](#), [15](#), [17](#), [18](#), [21](#), [23](#), [25–27](#)
- count, CppMap-method (count), [11](#)
- count, CppMultimap-method (count), [11](#)
- count, CppMultiset-method (count), [11](#)
- count, CppSet-method (count), [11](#)
- count, CppUnorderedMap-method (count), [11](#)
- count, CppUnorderedMultimap-method (count), [11](#)
- count, CppUnorderedMultiset-method (count), [11](#)
- count, CppUnorderedSet-method (count), [11](#)
- cpp\_deque, [12](#), [14](#), [15](#), [28](#)
- cpp\_forward\_list, [13](#), [13](#), [15](#), [28](#)
- cpp\_list, [13](#), [14](#), [14](#), [28](#)
- cpp\_map, [15](#), [17](#), [24](#), [25](#)
- cpp\_multimap, [16](#), [16](#), [24](#), [25](#)
- cpp\_multiset, [17](#), [21](#), [26](#), [27](#)
- cpp\_priority\_queue, [19](#), [23](#), [58](#)
- cpp\_queue, [19](#), [20](#), [23](#)
- cpp\_set, [18](#), [21](#), [26](#), [27](#)
- cpp\_stack, [19](#), [22](#)
- cpp\_unordered\_map, [16](#), [17](#), [23](#), [25](#)
- cpp\_unordered\_multimap, [16](#), [17](#), [24](#), [24](#)
- cpp\_unordered\_multiset, [18](#), [21](#), [25](#), [27](#)
- cpp\_unordered\_set, [18](#), [21](#), [26](#), [27](#)
- cpp\_vector, [13–15](#), [28](#), [36](#)
- emplace, [6](#), [12](#), [14](#), [15](#), [17–23](#), [25–28](#), [29](#), [31](#), [32](#), [38](#), [39](#), [44](#), [45](#), [49](#), [61](#), [63](#), [64](#)
- emplace, CppDeque-method (emplace), [29](#)
- emplace, CppList-method (emplace), [29](#)
- emplace, CppMap-method (emplace), [29](#)
- emplace, CppMultimap-method (emplace), [29](#)
- emplace, CppMultiset-method (emplace), [29](#)
- emplace, CppPriorityQueue-method (emplace), [29](#)
- emplace, CppQueue-method (emplace), [29](#)
- emplace, CppSet-method (emplace), [29](#)
- emplace, CppStack-method (emplace), [29](#)
- emplace, CppUnorderedMap-method (emplace), [29](#)
- emplace, CppUnorderedMultimap-method (emplace), [29](#)
- emplace, CppUnorderedMultiset-method (emplace), [29](#)
- emplace, CppUnorderedSet-method (emplace), [29](#)
- emplace, CppVector-method (emplace), [29](#)
- emplace\_after, [6](#), [13](#), [30](#), [30](#), [32](#), [39](#), [64](#)
- emplace\_after, CppForwardList-method (emplace\_after), [30](#)
- emplace\_back, [6](#), [7](#), [12](#), [14](#), [28](#), [30](#), [31](#), [31](#), [32](#), [46](#), [50](#), [64](#)
- emplace\_back, CppDeque-method (emplace\_back), [31](#)
- emplace\_back, CppList-method (emplace\_back), [31](#)
- emplace\_back, CppVector-method (emplace\_back), [31](#)
- emplace\_front, [6](#), [12–14](#), [30–32](#), [32](#), [37](#), [47](#), [51](#), [64](#)
- emplace\_front, CppDeque-method (emplace\_front), [32](#)
- emplace\_front, CppForwardList-method (emplace\_front), [32](#)
- emplace\_front, CppList-method (emplace\_front), [32](#)
- empty, [10](#), [12–15](#), [17–23](#), [25–28](#), [33](#), [34](#), [35](#), [52](#)
- empty, CppDeque-method (empty), [33](#)
- empty, CppForwardList-method (empty), [33](#)
- empty, CppList-method (empty), [33](#)
- empty, CppMap-method (empty), [33](#)
- empty, CppMultimap-method (empty), [33](#)
- empty, CppMultiset-method (empty), [33](#)
- empty, CppPriorityQueue-method (empty), [33](#)
- empty, CppQueue-method (empty), [33](#)
- empty, CppSet-method (empty), [33](#)
- empty, CppStack-method (empty), [33](#)
- empty, CppUnorderedMap-method (empty), [33](#)
- empty, CppUnorderedMultimap-method

- (empty), 33
- empty, CppUnorderedMultiset-method (empty), 33
- empty, CppUnorderedSet-method (empty), 33
- empty, CppVector-method (empty), 33
- erase, 10, 12, 14, 15, 17, 18, 21, 23, 25–28, 34, 35, 52, 65
- erase, CppDeque-method (erase), 34
- erase, CppList-method (erase), 34
- erase, CppMap-method (erase), 34
- erase, CppMultimap-method (erase), 34
- erase, CppMultiset-method (erase), 34
- erase, CppSet-method (erase), 34
- erase, CppUnorderedMap-method (erase), 34
- erase, CppUnorderedMultimap-method (erase), 34
- erase, CppUnorderedMultiset-method (erase), 34
- erase, CppUnorderedSet-method (erase), 34
- erase, CppVector-method (erase), 34
- erase\_after, 13, 34, 35
- erase\_after, CppForwardList-method (erase\_after), 35
  
- flip, 28, 36
- flip, CppVector-method (flip), 36
- front, 6, 7, 11–14, 20, 28, 37, 45, 47, 49, 51, 67
- front, CppDeque-method (front), 37
- front, CppForwardList-method (front), 37
- front, CppList-method (front), 37
- front, CppQueue-method (front), 37
- front, CppVector-method (front), 37
  
- insert, 6, 12, 14, 15, 17, 18, 21, 23, 25–28, 30–32, 38, 39, 40, 44, 50, 51, 64
- insert, CppDeque-method (insert), 38
- insert, CppList-method (insert), 38
- insert, CppMap-method (insert), 38
- insert, CppMultimap-method (insert), 38
- insert, CppMultiset-method (insert), 38
- insert, CppSet-method (insert), 38
- insert, CppUnorderedMap-method (insert), 38
- insert, CppUnorderedMultimap-method (insert), 38
- insert, CppUnorderedMultiset-method (insert), 38
- insert, CppUnorderedSet-method (insert), 38
- insert, CppVector-method (insert), 38
- insert\_after, 6, 13, 38, 39, 40
- insert\_after, CppForwardList-method (insert\_after), 39
- insert\_or\_assign, 6, 15, 23, 29, 38, 39, 40, 63, 64
- insert\_or\_assign, CppMap-method (insert\_or\_assign), 40
- insert\_or\_assign, CppUnorderedMap-method (insert\_or\_assign), 40
  
- load\_factor, 8, 23, 25–27, 41, 42, 51, 53, 57
- load\_factor, CppUnorderedMap-method (load\_factor), 41
- load\_factor, CppUnorderedMultimap-method (load\_factor), 41
- load\_factor, CppUnorderedMultiset-method (load\_factor), 41
- load\_factor, CppUnorderedSet-method (load\_factor), 41
  
- max\_bucket\_count, 8, 23, 25–27, 41, 41, 42, 43, 51, 53
- max\_bucket\_count, CppUnorderedMap-method (max\_bucket\_count), 41
- max\_bucket\_count, CppUnorderedMultimap-method (max\_bucket\_count), 41
- max\_bucket\_count, CppUnorderedMultiset-method (max\_bucket\_count), 41
- max\_bucket\_count, CppUnorderedSet-method (max\_bucket\_count), 41
- max\_load\_factor, 23, 25–27, 41, 42, 42, 43, 51, 53
- max\_load\_factor, CppUnorderedMap-method (max\_load\_factor), 42
- max\_load\_factor, CppUnorderedMultimap-method (max\_load\_factor), 42
- max\_load\_factor, CppUnorderedMultiset-method (max\_load\_factor), 42
- max\_load\_factor, CppUnorderedSet-method (max\_load\_factor), 42
- max\_size, 12–15, 17, 18, 21, 23, 25–28, 43, 57
- max\_size, CppDeque-method (max\_size), 43
- max\_size, CppForwardList-method (max\_size), 43
- max\_size, CppList-method (max\_size), 43
- max\_size, CppMap-method (max\_size), 43

- max\_size, CppMultimap-method (max\_size), 43
- max\_size, CppMultiset-method (max\_size), 43
- max\_size, CppSet-method (max\_size), 43
- max\_size, CppUnorderedMap-method (max\_size), 43
- max\_size, CppUnorderedMultimap-method (max\_size), 43
- max\_size, CppUnorderedMultiset-method (max\_size), 43
- max\_size, CppUnorderedSet-method (max\_size), 43
- max\_size, CppVector-method (max\_size), 43
- merge, 14, 15, 17, 18, 21, 23, 25–27, 44, 59, 60
- merge, CppForwardList, CppForwardList-method (merge), 44
- merge, CppList, CppList-method (merge), 44
- merge, CppMap, CppMap-method (merge), 44
- merge, CppMultimap, CppMultimap-method (merge), 44
- merge, CppMultiset, CppMultiset-method (merge), 44
- merge, CppSet, CppSet-method (merge), 44
- merge, CppUnorderedMap, CppUnorderedMap-method (merge), 44
- merge, CppUnorderedMultimap, CppUnorderedMultimap-method (merge), 44
- merge, CppUnorderedMultiset, CppUnorderedMultiset-method (merge), 44
- merge, CppUnorderedSet, CppUnorderedSet-method (merge), 44
  
- pop, 19, 20, 22, 37, 45, 46, 47, 49, 61
- pop, CppPriorityQueue-method (pop), 45
- pop, CppQueue-method (pop), 45
- pop, CppStack-method (pop), 45
- pop\_back, 7, 12, 14, 28, 32, 46, 47, 50
- pop\_back, CppDeque-method (pop\_back), 46
- pop\_back, CppList-method (pop\_back), 46
- pop\_back, CppVector-method (pop\_back), 46
- pop\_front, 12–14, 32, 37, 46, 47, 51
- pop\_front, CppDeque-method (pop\_front), 47
- pop\_front, CppForwardList-method (pop\_front), 47
- pop\_front, CppList-method (pop\_front), 47
- print, 12–15, 17–23, 25–28, 47, 62, 64
- print, CppDeque-method (print), 47
- print, CppForwardList-method (print), 47
- print, CppList-method (print), 47
- print, CppMap-method (print), 47
- print, CppMultimap-method (print), 47
- print, CppMultiset-method (print), 47
- print, CppPriorityQueue-method (print), 47
- print, CppQueue-method (print), 47
- print, CppSet-method (print), 47
- print, CppStack-method (print), 47
- print, CppUnorderedMap-method (print), 47
- print, CppUnorderedMultimap-method (print), 47
- print, CppUnorderedMultiset-method (print), 47
- print, CppUnorderedSet-method (print), 47
- print, CppVector-method (print), 47
- push, 19, 20, 22, 38, 45, 49, 49, 61
- push, CppPriorityQueue-method (push), 49
- push, CppQueue-method (push), 49
- push, CppStack-method (push), 49
- push\_back, 7, 12, 14, 28, 32, 46, 50, 51
- push\_back, CppDeque-method (push\_back), 50
- push\_back, CppList-method (push\_back), 50
- push\_back, CppVector-method (push\_back), 50
- push\_front, 12–14, 32, 37, 47, 50, 50
- push\_front, CppDeque-method (push\_front), 50
- push\_front, CppForwardList-method (push\_front), 50
- push\_front, CppList-method (push\_front), 50
  
- rehash, 23, 25–27, 51
- rehash, CppUnorderedMap-method (rehash), 51
- rehash, CppUnorderedMultimap-method (rehash), 51
- rehash, CppUnorderedMultiset-method (rehash), 51
- rehash, CppUnorderedSet-method (rehash), 51
  
- remove., 10, 13, 14, 34, 35, 52, 65
- remove., CppForwardList-method (remove.), 52
- remove., CppList-method (remove.), 52
- reserve, 9, 23, 25–28, 51, 53, 56

- reserve, CppUnorderedMap-method (reserve), 53
- reserve, CppUnorderedMultimap-method (reserve), 53
- reserve, CppUnorderedMultiset-method (reserve), 53
- reserve, CppUnorderedSet-method (reserve), 53
- reserve, CppVector-method (reserve), 53
- resize, 12–14, 28, 54, 57
- resize, CppDeque-method (resize), 54
- resize, CppForwardList-method (resize), 54
- resize, CppList-method (resize), 54
- resize, CppMap-method (resize), 54
- resize, CppMultimap-method (resize), 54
- resize, CppMultiset-method (resize), 54
- resize, CppPriorityQueue-method (resize), 54
- resize, CppQueue-method (resize), 54
- resize, CppSet-method (resize), 54
- resize, CppStack-method (resize), 54
- resize, CppUnorderedMap-method (resize), 54
- resize, CppUnorderedMultimap-method (resize), 54
- resize, CppUnorderedMultiset-method (resize), 54
- resize, CppUnorderedSet-method (resize), 54
- resize, CppVector-method (resize), 54
- reverse, 13, 14, 55
- reverse, CppForwardList-method (reverse), 55
- reverse, CppList-method (reverse), 55
- shrink\_to\_fit, 9, 12, 28, 55
- shrink\_to\_fit, CppDeque-method (shrink\_to\_fit), 55
- shrink\_to\_fit, CppVector-method (shrink\_to\_fit), 55
- size, 8, 9, 12, 14, 15, 17–23, 25–28, 43, 56, 56
- size, CppDeque-method (size), 56
- size, CppList-method (size), 56
- size, CppMap-method (size), 56
- size, CppMultimap-method (size), 56
- size, CppMultiset-method (size), 56
- size, CppPriorityQueue-method (size), 56
- size, CppQueue-method (size), 56
- size, CppSet-method (size), 56
- size, CppStack-method (size), 56
- size, CppUnorderedMap-method (size), 56
- size, CppUnorderedMultimap-method (size), 56
- size, CppUnorderedMultiset-method (size), 56
- size, CppUnorderedSet-method (size), 56
- size, CppVector-method (size), 56
- sort, 13, 14, 57, 58, 65
- sort, CppForwardList-method (sort), 57
- sort, CppList-method (sort), 57
- sorting, 5, 19, 48, 57, 58, 62, 64
- sorting, CppPriorityQueue-method (sorting), 58
- splice, 14, 59, 60
- splice, CppList-method (splice), 59
- splice\_after, 13, 59, 60
- splice\_after, CppForwardList-method (splice\_after), 60
- to\_r, 12–15, 17–23, 25–28, 48, 61, 64
- to\_r, CppDeque-method (to\_r), 61
- to\_r, CppForwardList-method (to\_r), 61
- to\_r, CppList-method (to\_r), 61
- to\_r, CppMap-method (to\_r), 61
- to\_r, CppMultimap-method (to\_r), 61
- to\_r, CppMultiset-method (to\_r), 61
- to\_r, CppPriorityQueue-method (to\_r), 61
- to\_r, CppQueue-method (to\_r), 61
- to\_r, CppSet-method (to\_r), 61
- to\_r, CppStack-method (to\_r), 61
- to\_r, CppUnorderedMap-method (to\_r), 61
- to\_r, CppUnorderedMultimap-method (to\_r), 61
- to\_r, CppUnorderedMultiset-method (to\_r), 61
- to\_r, CppUnorderedSet-method (to\_r), 61
- to\_r, CppVector-method (to\_r), 61
- top, 6, 7, 11, 19, 22, 45, 49, 61, 67
- top, CppPriorityQueue-method (top), 61
- top, CppStack-method (top), 61
- try\_emplace, 15, 23, 30, 63
- try\_emplace, CppMap-method (try\_emplace), 63
- try\_emplace, CppUnorderedMap-method (try\_emplace), 63
- type, 5, 12–15, 17–23, 25–28, 48, 62, 64
- type, CppDeque-method (type), 64



type, CppForwardList-method (type), 64  
type, CppList-method (type), 64  
type, CppMap-method (type), 64  
type, CppMultimap-method (type), 64  
type, CppMultiset-method (type), 64  
type, CppPriorityQueue-method (type), 64  
type, CppQueue-method (type), 64  
type, CppSet-method (type), 64  
type, CppStack-method (type), 64  
type, CppUnorderedMap-method (type), 64  
type, CppUnorderedMultimap-method  
(type), 64  
type, CppUnorderedMultiset-method  
(type), 64  
type, CppUnorderedSet-method (type), 64  
type, CppVector-method (type), 64  
  
unique, 13, 14, 57, 65  
unique, CppForwardList-method (unique),  
65  
unique, CppList-method (unique), 65