Package: cppcontainers (via r-universe)

February 7, 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 Düben [aut, cre]

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

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

Additional_repositories https://cranhaven.r-universe.dev

Repository https://cranhaven.r-universe.dev

RemoteUrl https://github.com/cranhaven/cranhaven.r-universe.dev

RemoteRef package/cppcontainers

RemoteSha 884e6be0128a5572fdcc8cd88409dba55c63a1e3

RemoteSubdir cppcontainers

Contents

`==,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` *Get container capacity*

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 *Count element frequency*

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 *Create forward list*

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

keys	An integer, numeric, character, or logical vector.
values	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` *Add an element*

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

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

---

emplace_back *Add an element to the back*

Description

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

Usage

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

Arguments

x	A CppVector, CppDeque, 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_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` *Toggle boolean values*

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 *Access first element*

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` *Add elements*

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 *Get the mean number of elements per bucket*

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 *Get the maximum number of buckets*

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` *Merge two objects*

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 *Remove an element from the back*

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 *Remove an element from the front*

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 *Print container data*

Description

Print the data in a container.

Usage

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

Arguments

x	A cppcontainers object.
...	An ellipsis for compatibility with the generic method. Accepts the parameters n, from, and to. See to_r for their effects. A difference to to_r 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 *Add elements*

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

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

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` *Reserve space*

Description

Reserve space for the container by reference.

Usage

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

Arguments

`x`	A CppUnorderedSet, CppUnorderedMultiset, CppUnorderedMap, CppUnordered-Multimap, 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

`x`	A `cppcontainers` object.
`size`	The new size of the container.
`value`	The value of new elements. It defaults to `0` for integers and doubles, to `""` for strings, and to `FALSE` 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 *Reverse element order*

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 *Shrink container capacity to size*

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 *Move elements*

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

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

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 *Access top element*

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 *Export data to R*

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 CppUnordered-Multimap 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