**Deques**

It manages its elements with a dynamic array, provides random access, and has almost the same interface as a vector.

Difference: Dynamic array is open at both ends

header file

#include <deque>

The type is defined as a class template inside namespace std:

namespace std {

template <typename T,

typename Allocator = allocator<T> >

class deque;

}

**T:** type of the elements

**Allocator:** memory model (default allocator)

# Abilities of Deques

**Difference from vector:**

* Inserting and removing elements is fast at both ends.
* With deques, element access and iterator movement are usually a bit slower.
* A deque might contain more elements because it uses more than one block of memory. Thus, max\_size() might be larger for deques.
* Deques provide no support to control the capacity and the moment of reallocation.
* Blocks of memory might get freed when they are no longer used, so the memory size of a deque might shrink.

**Similarity with vector:**

* Inserting and deleting elements in the middle is relatively slow because all elements up to either end may be moved to make room or to fill a gap.
* Iterators are random-access iterators.

**Prefer a deque if the following are true:**

* You insert and remove elements at both ends (this is the classic case for a queue).
* You don’t refer to elements of the container.
* It is important that the container frees memory when it is no longer used (however, the standard does not guarantee that this happens).

# Deque Operations

## Create, Copy and Destroy

|  |  |
| --- | --- |
| Operation | Effect |
| deque<Elem> c | Default constructor; creates an empty deque without any elements |
| deque<Elem> c(c2) | Copy constructor; creates a new deque as a copy of c2 (all elements are copied) |
| deque<Elem> c = c2 | Copy constructor; creates a new deque as a copy of c2 (all elements are copied) |
| deque<Elem> c(rv) | Move constructor; creates a new deque, taking the contents of the rvalue rv (since C++11) |
| deque<Elem> c = rv | Move constructor; creates a new deque, taking the contents of the rvalue rv (since C++11) |
| deque<Elem> c(n) | Creates a deque with n elements created by the default constructor |
| deque<Elem> c(n,elem) | Creates a deque initialized with n copies of element elem |
| deque<Elem> c(beg,end) | Creates a deque initialized with the elements of the range [beg,end) |
| deque<Elem> c(initlist) | Creates a deque initialized with the elements of initializer list initlist (since C++11) |
| deque<Elem> c = initlist | Creates a deque initialized with the elements of initializer list initlist (since C++11) |
| c.~deque() | Destroys all elements and frees the memory |

Deque operations differ from vector operations in only two ways:

1. Deques do not provide the functions for capacity (capacity() and reserve()).
2. Deques do provide direct functions to insert and to delete the first element (push\_front() and pop\_front()).

Other operations are the same.

### shrink\_to\_fit()

* added with C++11
* nonbinding request to shrink the internal memory to fit the number of elements
* **Why shrink\_to\_fit() if deques are allowed to free blocks of memory ?**

The memory that contains all the pointers to the blocks of memory usually does not shrink, which might change with this call.

## Nonmodifying Operations

|  |  |
| --- | --- |
| Operation | Effect |
| c.empty() | Returns whether the container is empty (equivalent to size()==0 but might be faster) |
| c.size() | Returns the current number of elements |
| c.max\_size() | Returns the maximum number of elements possible |
| c.shrink\_to\_fit() | Request to reduce capacity to fit number of elements (since C++11) |
| c1 == c2 | Returns whether c1 is equal to c2 (calls == for the elements) |
| c1 != c2 | Returns whether c1 is not equal to c2 (equivalent to !(c1==c2)) |
| c1 < c2 | Returns whether c1 is less than c2 |
| c1 > c2 | Returns whether c1 is greater than c2 (equivalent to c2<c1) |
| c1 <= c2 | Returns whether c1 is less than or equal to c2 (equivalent to !(c2<c1)) |
| c1 >= c2 | Returns whether c1 is greater than or equal to c2 (equivalent to !(c1<c2)) |

shrink\_to\_fit() manipulates the deque because it invalidates references, pointers, and iterators to elements. However, it is listed as a nonmodifying operation because it does not manipulate the logical contents of the container.

## Element Access

|  |  |
| --- | --- |
| Operation | Effect |
| c[idx] | Returns the element with index idx (no range checking) |
| c.at(idx) | Returns the element with index idx (throws range-error exception if idx is out of range) |
| c.front() | Returns the first element (no check whether a first element exists) |
| c.back() | Returns the last element (no check whether a last element exists) |

* No member functions for element access (except at()) check whether an index or an iterator is valid.
* An insertion or deletion of elements might cause a reallocation.
* Thus, any insertion or deletion invalidates all pointers, references, and iterators that refer to other elements of the deque.
* The exception is when elements are inserted at the front or the back. In this case, references and pointers to elements stay valid, but iterators don’t.

## Iterator Functions

|  |  |
| --- | --- |
| Operation | Effect |
| c.begin() | Returns a random-access iterator for the first element |
| c.end() | Returns a random-access iterator for the position after the last element |
| c.cbegin() | Returns a constant random-access iterator for the first element (since C++11) |
| c.cend() | Returns a constant random-access iterator for the position after the last element (since C++11) |
| c.rbegin() | Returns a reverse iterator for the first element of a reverse iteration |
| c.rend() | Returns a reverse iterator for the position after the last element of a reverse iteration |
| c.crbegin() | Returns a constant reverse iterator for the first element of a reverse iteration (since C++11) |
| c.crend() | Returns a constant reverse iterator for the position after the last element of a reverse iteration (since C++11) |

## Inserting and removing elements

|  |  |
| --- | --- |
| Operation | Effect |
| c = c2 | Assigns all elements of c2 to c |
| c = rv | Move assigns all elements of the rvalue rv to c (since C++11) |
| c = initlist | Assigns all elements of the initializer list initlist to c (since C++11) |
| c.assign(n, elem) | Assigns n copies of element elem |
| c.assign(beg, end) | Assigns the elements of the range [beg,end) |
| c.assign(initlist) | Assigns all the elements of the initializer list initlist |
| c1.swap(c2) | Swaps the data of c1 and c2 |
| swap(c1, c2) | Swaps the data of c1 and c2 |
| c.push\_back(elem) | Appends a copy of elem at the end |
| c.pop\_back() | Removes the last element (does not return it) |
| c.push\_front(elem) | Inserts a copy of elem at the beginning |
| c.pop\_front() | Removes the first element (does not return it) |
| c.insert(pos, elem) | Inserts a copy of elem before iterator position pos and returns the position of the new element |
| c.insert(pos, n, elem) | Inserts n copies of elem before iterator position pos and returns the position of the first new element (or pos if there is no new element) |
| c.insert(pos, beg, end) | Inserts a copy of all elements of the range [beg,end) before iterator position pos and returns the position of the first new element (or pos if there is no new element) |
| c.insert(pos, initlist) | Inserts a copy of all elements of the initializer list initlist before iterator position pos and returns the position of the first new element (or pos if there is no new element; since C++11) |
| c.emplace(pos, args...) | Inserts a copy of an element initialized with args before iterator position pos and returns the position of the new element (since C++11) |
| c.emplace\_back(args...) | Appends a copy of an element initialized with args at the end (returns nothing; since C++11) |
| c.emplace\_front(args...) | Inserts a copy of an element initialized with args at the beginning (returns nothing; since C++11) |
| c.erase(pos) | Removes the element at iterator position pos and returns the position of the next element |
| c.erase(beg, end) | Removes all elements of the range [beg,end) and returns the position of the next element |
| c.resize(num) | Changes the number of elements to num (if size() grows new elements are created by their default constructor) |
| c.resize(num, elem) | Changes the number of elements to num (if size() grows new elements are copies of elem) |
| c.clear() | Removes all elements (empties the container) |

### resize

* If n is smaller than the current container size, the content is reduced to its first n elements, removing those beyond (and destroying them).

A call to resize with a smaller size does not invalidate any references to non-erased elements.

* If n is greater than the current container size, the content is expanded by inserting at the end as many elements as needed to reach a size of n. If val is specified, the new elements are initialized as copies of val, otherwise, they are value-initialized.

A call to resize with a bigger size does not invalidate any references to elements of the deque.

# Exception Handling

Deques provide the same support for exception handing that vectors do.

Additional operations push\_front() and pop\_front() behave according to push\_back() and pop\_back(), respectively.

* If an element gets inserted with push\_back() or push\_front() and an exception occurs, these functions have no effect.
* Neither pop\_back() nor pop\_front() throws any exceptions.
* Member function at() throws out\_of\_range if n is out of bounds.

If an exception is thrown, there are no changes in the container.

# Example

#include <iostream>

#include <deque>

#include <string>

#include <algorithm>

#include <iterator>

using namespace std;

int main() {

// create empty deque of strings

deque<string> coll;

// insert several elements

coll.assign (3, string("string"));

coll.push\_back ("last string");

coll.push\_front ("first string");

// print elements separated by newlines

copy (coll.cbegin(), coll.cend(), ostream\_iterator<string>(cout,", "));

cout << endl;

// remove first and last element

coll.pop\_front();

coll.pop\_back();

// insert ‘‘another’’ into every element but the first

for (unsigned i=1; i<coll.size(); ++i) {

coll[i] = "another " + coll[i];

}

copy (coll.cbegin(), coll.cend(), ostream\_iterator<string>(cout,", "));

cout << endl;

// change size to four elements

coll.resize (4, "resized string");

// print elements separated by newlines

copy (coll.cbegin(), coll.cend(), ostream\_iterator<string>(cout,", "));

cout << endl;

return 0;

}

Output:

first string, string, string, string, last string,

string, another string, another string,

string, another string, another string, resized string,

# END