
7.4. Deques

A deque (pronounced “deck”) is very similar to a vector. It manages its elements with a dynamic array, provides random access, and has almost the same interface as a vector. The difference is that with a deque, the dynamic array is open at both ends. Thus, a deque is fast for insertions and deletions at both the end and the beginning (Figure 7.3).

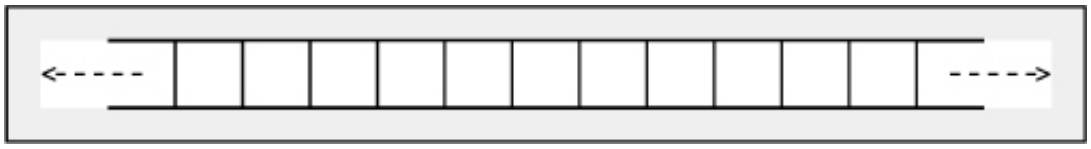

Figure 7.3. Logical Structure of a Deque

To provide this ability, the deque is typically implemented as a bunch of individual blocks, with the first block growing in one direction and the last block growing in the opposite direction (Figure 7.4).

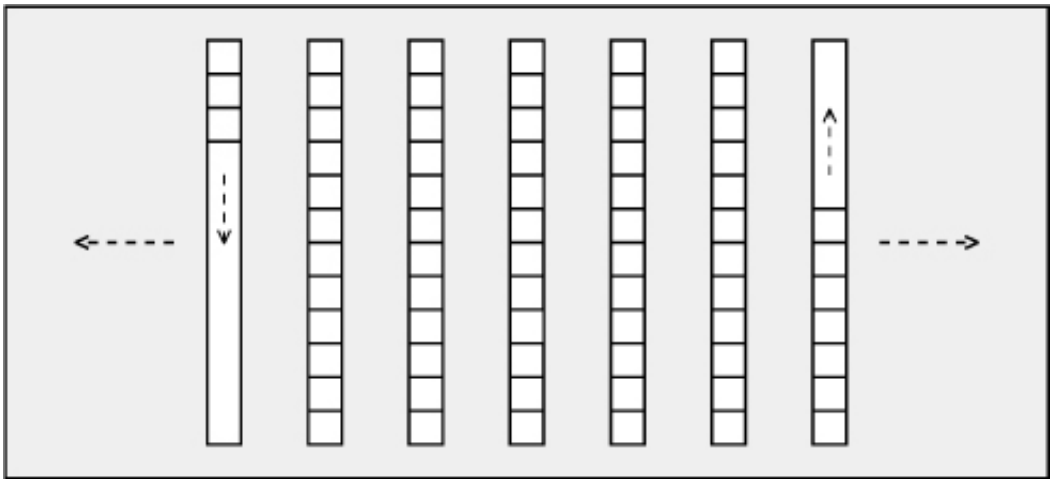

Figure 7.4. Internal Structure of a Deque

To use a deque, you must include the header file `<deque>`:

```
#include <deque>
```

There, the type is defined as a class template inside namespace `std`:

```
namespace std {
    template <typename T,
              typename Allocator = allocator<T> >
    class deque;
}
```

As with all sequence containers, the type of the elements is passed as a first template parameter. The optional second template argument is the memory model, with `allocator` as the default (see Chapter 19).

7.4.1. Abilities of Deques

The abilities of deques differ from those of vectors as follows:

- Inserting and removing elements is fast at both the beginning and the end (for vectors, it is fast only at the end). These operations are done in amortized constant time.
- The internal structure has one more indirection to access the elements, so with deques, element access and iterator movement are usually a bit slower.
- Iterators must be smart pointers of a special type rather than ordinary pointers because they must jump between different blocks.
- In systems that have size limitations for blocks of memory (for example, some PC systems), 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. In particular, any insertion or deletion of elements other than at the beginning or end invalidates all pointers, references, and iterators that refer to elements of the deque. However, reallocation may perform better than for vectors because according to their typical internal structure, deques don’t have to copy all elements on reallocation.
- Blocks of memory might get freed when they are no longer used, so the memory size of a deque might shrink (however, whether

and how this happens is implementation specific).

The following features of vectors apply also to deques:

- 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.

In summary, you should 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).

The interface of vectors and deques is almost the same, so trying both is very easy when no special feature of a vector or a deque is necessary.

7.4.2. Deque Operations

Tables 7.16 through 7.18 list all operations provided for deques.[8]

[8] `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.

Table 7.16. Constructors and Destructor of Deques

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

Table 7.17. Nonmodifying Operations of Deques

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)[8]
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*`)`)
`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)
`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)

Table 7.18. Modifying Operations of Deques

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 new 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 new element initialized with *args* at the end (returns nothing; since C++11)
`c.emplace_front(args...)`	Inserts a new 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)

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()`).

Because the other operations are the same, they are not explained again here. See Section 7.3.2, page 273, for a description of them.

Note, however, that `shrink_to_fit()` was added with C++11 as nonbinding request to shrink the internal memory to fit the number of elements. You might argue that `shrink_to_fit()` makes no sense for deques because they are allowed to free blocks of memory. However, the memory that contains all the pointers to the blocks of memory usually does not shrink, which might change with this call.

In addition, note that you still must consider the following:

1. No member functions for element access (except `at()`) check whether an index or an iterator is valid.

2. 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.

7.4.3. Exception Handling

In principle, deques provide the same support for exception handing that vectors do (see Section 7.3.4, page 278). The additional operations `push_front()`, `emplace_front()`, and `pop_front()` behave according to `push_back()`, `emplace_back()`, and `pop_back()`, respectively. Thus, the C++ standard library provides the following behavior:

- If an element gets inserted with `push_front()`, `emplace_front()`, `push_back()`, or `emplace_back()` and an exception occurs, these functions have no effect. Again, move operations for the elements that guarantee not to throw will improve the performance.
- Neither `pop_back()` nor `pop_front()` throws any exceptions.

See Section 6.12.2, page 248, for a general discussion of exception handling in the STL.

7.4.4. Examples of Using Deques

The following program shows the abilities of deques:

```
// cont/deque1.cpp

#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,"\n"));
    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];
    }

    // change size to four elements
    coll.resize (4, "resized string");

    // print elements separated by newlines
    copy (coll.cbegin(), coll.cend(),
          ostream_iterator<string>(cout,"\n"));
}
```

The program has the following output:

```
first string
string
string
string
last string

string
another string
another string
resized string
```