
7.1. Common Container Abilities and Operations

7.1.1. Container Abilities

This section covers the common abilities of STL container classes. Most of these abilities are requirements that, in general, every STL container should meet. The three core abilities are as follows:

1. All containers provide value rather than reference semantics. Containers copy and/or move elements internally when they are inserted rather than managing references to them. Thus, ideally, each element of an STL container must be able to be copied and moved. If objects you want to store don't have a public copy constructor, or if copying is not useful — because, for example, it takes too much time or elements must be part of multiple containers — you might use only move operations, or the container elements must be pointers or pointer objects that refer to these objects. Section 7.11, page 388, provides an example for using shared pointers to get reference semantics.
2. The elements inside a container have a specific order. Each container type provides operations that return iterators to iterate over the elements. This is the key interface of the STL algorithms. Thus, if you iterate multiple times over the elements, you will find the same order, provided that you don't insert or delete elements. This applies even to "unordered containers," as long as you don't call operations that add or delete elements or force an internal reorganization.
3. In general, operations are not "safe" in the sense that they check for every possible error. The caller must ensure that the parameters of the operations meet the requirements these operations have. Violating these requirements, such as using an invalid index, results in undefined behavior, which means that anything can happen.

   Usually, the STL does *not* throw exceptions by itself. If user-defined operations called by the STL containers do throw, the behavior differs. See Section 6.12.2, page 248, for details.

7.1.2. Container Operations

The standard specifies a list of common container requirements that shall apply to all STL containers. However, due to the variety of containers provided with C++11, there might be exceptions so that some containers don't even fulfill all general container requirements, and that there are additional operations provided by all containers. Tables 7.1 and 7.2 list the operations that are common to (almost) all containers. Column "**Req**" signs operations that are part of the general container requirements. The following subsections explore some of these common operations.

Table 7.1. Common Operations of (Almost All) Container Classes, Part 1

Operation	Req	Effect
ContType `c`	Yes	Default constructor; creates an empty container without any element (`array<>` gets default elements)
ContType `c(`*c2*`)`	Yes	Copy constructor; creates a new container as a copy of *c2* (all elements are copied)
ContType `c =` *c2*	Yes	Copy constructor; creates a new container as a copy of *c2* (all elements are copied)
ContType `c(`*rv*`)`	Yes	Move constructor; creates a new container, taking the contents of the rvalue *rv* (since C++11; not for `array<>`)
ContType `c =` *rv*	Yes	Move constructor; creates a new container, taking the contents of the rvalue *rv* (since C++11; not for `array<>`)
ContType `c(`*beg*`,`*end*`)`	–	Creates a container and initializes it with copies of all elements of [*beg*,*end*) (not for `array<>`)
ContType `c(`*initlist*`)`	–	Creates a container and initializes it with copies of the values of the initializer list *initlist* (since C++11; not for `array<>`)
ContType `c =` *initlist*	–	Creates a container and initializes it with copies of the values of the initializer list *initlist* (since C++11)
`c.~`*ContType*`()`	Yes	Deletes all elements and frees the memory, if possible
`c.empty()`	Yes	Returns whether the container is empty (equivalent to `size()==0` but might be faster)
`c.size()`	Yes	Returns the current number of elements (not for `forward_list<>`)
`c.max_size()`	Yes	Returns the maximum number of elements possible

`c1 == c2`	Yes	Returns whether *c1* is equal to *c2*
`c1 != c2`	Yes	Returns whether *c1* is not equal to *c2* (equivalent to `!(c1==c2)`)
`c1 < c2`	–	Returns whether *c1* is less than *c2* (not for unordered containers)
`c1 > c2`	–	Returns whether *c1* is greater than *c2* (equivalent to *c2*<*c1*; not for unordered containers)
`c1 <= c2`	–	Returns whether *c1* is less than or equal to *c2* (equivalent to `!(c2<c1)`; not for unordered containers)
`c1 >= c2`	–	Returns whether *c1* is greater than or equal to *c2* (equivalent to `!(c1<c2)`; not for unordered containers)
`c = c2`	Yes	Assigns all elements of *c2* to *c*
`c = rv`	Yes	Move assigns all elements of the rvalue *rv* to *c* (since C++11; not for `array<>`)
`c = initlist`	–	Assigns all elements of the initializer list *initlist* (since C++11; not for `array<>`)
`c1.swap(c2)`	Yes	Swaps the data of *c1* and *c2*
`swap(c1,c2)`	Yes	Swaps the data of *c1* and *c2*

Table 7.2. Common Operations of (Almost All) Container Classes, Part 2

Operation	Req	Effect
`c.begin()`	Yes	Returns an iterator for the first element
`c.end()`	Yes	Returns an iterator for the position after the last element
`c.cbegin()`	Yes	Returns a constant iterator for the first element (since C++11)
`c.cend()`	Yes	Returns a constant iterator for the position after the last element (since C++11)
`c.clear()`	–	Removes all elements (empties the container; not for `array<>`)

Initialization

Every container class provides a default constructor, a copy constructor, and a destructor. You can also initialize a container with elements of a given range and, since C++11, with an initializer list.

The constructor for an initializer list (see Section 3.1.3, page 15) provides a convenient way to specify initial values. This is especially useful to initialize constant containers:

Click here to view code image

```
// initialize a vector with some specific values (since C++11)
const std::vector<int> v1 = { 1, 2, 3, 5, 7, 11, 13, 17, 21 };

// same with different syntax
const std::vector<int> v2 { 1, 2, 3, 5, 7, 11, 13, 17, 21 };

// initialize an unordered set with "hello" and two empty strings
std::unordered_set<std::string> w = { "hello", std::string(), "" };
```

Some special rules apply to the use of initializer lists for `array<>` containers (see Section 7.2.1, page 262, for details).

The constructor for a given range provides the ability to initialize the container with elements of another container, with a C-style array, or from an input stream. This constructor is a member template (see Section 3.2, page 34), so not only the container but also the type of the elements may differ, provided that there is an automatic conversion from the source element type to the destination element type. For example:

- You can initialize a container with the elements of another container:

```
std::list<int> l;          // l is a linked list of ints

// copy all elements of the list as floats into a vector
std::vector<float> c(l.begin(),l.end());
```

Since C++11, you can also move the elements here, using a move iterator (see Section 9.4.4, page 466):

Click here to view code image

```
std::list<std::string> l;      // l is a linked list of strings

// move all elements of the list into a vector
std::vector<std::string> c(std::make_move_iterator(l.begin()),
                           std::make_move_iterator(l.end()));
```

- You can initialize a container with the elements of an ordinary C-style array:

```
int carray[] = { 2, 3, 17, 33, 45, 77 };

// copy all elements of the C-style array into a set
std::set<int> c(std::begin(carray),std::end(carray));
```

`std::begin()` and `std::end()` for C-style arrays are defined since C++11 in `<iterator>`.

Note that before C++11, you had to call:

```
std::set<int> c(carray,carray+sizeof(carray)/sizeof(carray[0]));
```

- You can initialize a container from an input stream, such as standard input:

```
// read all integer elements of the deque from standard input
std::deque<int> c{std::istream_iterator<int>(std::cin),
                  std::istream_iterator<int>()};
```

Note that you should use the new uniform initialization syntax with brackets (see Section 3.1.3, page 15). Otherwise, you need extra parentheses around the initializer arguments here:

```
// read all integer elements of the deque from standard input
std::deque<int> c((std::istream_iterator<int>(std::cin)),
                  (std::istream_iterator<int>()));
```

The reason is that without the extra parentheses, you specify something very different, so you will probably get some strange warnings or errors in following statements. Consider writing the statement without extra parentheses:

```
std::deque<int> c(std::istream_iterator<int>(std::cin),
                  std::istream_iterator<int>());
```

In this case, `c` declares a *function* having `deque<int>` as return type. Its first parameter is of type `istream_iterator<int>` with the name `cin`, and its second unnamed parameter is of type "function taking no arguments returning `istream_iterator<int>`." This construct is valid syntactically as either a declaration or an expression. So, according to language rules, it is treated as a declaration. The extra parentheses force the initializer not to match the syntax of a declaration.[2]

[2] Thanks to John H. Spicer from EDG for this explanation.

In principle, these techniques are also provided to assign or to insert elements from another range. However, for those operations, the exact interfaces either differ due to additional arguments or are not provided for all container classes.

Finally, since C++11, you can use a move constructor (see Section 3.1.5, page 21) to initialize a container (for `array<>`, it is implicitly defined):

```
std::vector<int> v1;

// move contents of v1 into v2, state of v1 undefined afterward
std::vector<int> v2 = std::move(v1);
```

As a result, the newly created container has the elements of the container used for initialization, whereas the contents of the container used for the initialization is unspecified afterward. This constructor provides significant performance improvements because internally, the elements are moved by switching some pointers instead of copying element by element. So whenever you no longer need a container, which gets copied, you should use the move constructor.

Assignments and `swap()`

If you assign containers, you copy all elements of the source container and remove all old elements in the destination container. Thus, assignment of containers is relatively expensive.

Since C++11, you can use the move assignment semantics instead (see Section 3.1.5, page 21). All containers provide move assignment operators (`array<>` implicitly again), declared for rvalues, which internally just swap pointers to the memory of values rather than copying all values. The exact behavior is not specified, but the guarantee of constant complexity for this operation leads to an implementation like this. The C++ standard library simply specifies that after a move assignment, the container on the left-hand side of the assignment has the elements that the container on the right-hand side of the assignment had before. The contents of the container on the right-hand side are undefined afterward:

```
std::vector<int> v1;
std::vector<int> v2;

// move contents of v1 into v2, state of v1 undefined afterward
```

```
v2 = std::move(v1);
```

So, for performance reasons, you should use this way of assignment if after an assignment, the contents of the container on the right-hand side are no longer used.

In addition and since C++98, all containers provide a `swap()` member function to swap contents of two containers. In fact, it swaps only some internal pointers that refer to the data (elements, allocator, sorting criterion, if any). So, `swap()` is guaranteed to have only constant complexity, not the linear complexity of a copy assignment. Iterators and references to elements of a container follow swapped elements. So, after `swap()`, iterators and references still refer to the elements they referred to before, which, however, are in different containers then.

Note that for containers of type `array<>`, the behavior of `swap()` is slightly different. Because you can't internally just swap pointers, `swap()` has linear complexity, and iterators and references refer to the same container but different elements afterward.

Size Operations

For (almost) all container classes, three size operations are provided:

1. **empty()** returns whether the number of elements is zero (`begin()==end()`). You should prefer it over `size()==0`, because it might be implemented more efficiently than `size()`, and `size()` is not provided for forward lists.
2. **size()** returns the current number of elements of the container. This operation is not provided for `forward_list<>` because it couldn't have constant complexity there.
3. **max_size()** returns the maximum number of elements a container might contain. This value is implementation defined. For example, a vector typically contains all elements in a single block of memory, so there might be relevant restrictions on PCs. Otherwise, `max_size()` is usually the maximum value of the type of the index.

Comparisons

For all but unordered containers, the usual comparison operators `==`, `!=`, `<`, `<=`, `>`, and `>=` are defined according to the following three rules:

1. Both containers must have the same type.
2. Two containers are equal if their elements are equal and have the same order. To check equality of elements, operator `==` is used.
3. To check whether a container is less than another container, a lexicographical comparison is done (see Section 11.5.4, page 548).

For unordered containers, only the operators `==` and `!=` are defined. They return `true` when each element in one container has an equal element in the other container. The order doesn't matter (that's why they are unordered containers).

Because the operators `<`, `<=`, `>`, and `>=` are not provided for unordered container, only the operators `==` and `!=` are a common container requirement. Before C++11 all comparison operators were required. Since C++11 there is an table of "optional container requirements" that covers the remaining four comparison operators.

To compare containers with different types, you must use the comparing algorithms of Section 11.5.4, page 542.

Element Access

All containers provide an iterator interface, which means that range-based `for` loops are supported (see Section 3.1.4, page 17). Thus, the easiest way to get access to all elements since C++11 is as follows:

```
for (const auto& elem : coll) {
    std::cout << elem << std::endl;
}
```

To be able to manipulate the elements, you should skip the `const`:

```
for (auto& elem : coll) {
    elem = ...;
}
```

To operate with positions (for example, to be able to insert, delete, or move elements around), you can always use iterators yielded by `cbegin()` and `cend()` for read-only access:

```
for (auto pos=coll.cbegin(); pos!=coll.cend(); ++pos) {
    std::cout << *pos << std::endl;
}
```

and iterators yielded by `begin()` and `end()` for read/write access:

```
for (auto pos=coll.begin(); pos!=coll.end(); ++pos) {
    *pos = ...;
}
```

Before C++11, you had to, and still can, declare the type of the iterator explicitly for read access:

```
colltype::const_iterator pos;
for (pos=coll.begin(); pos!=coll.end(); ++pos) {
    ...;
}
```

or for read/write access:

```
colltype::iterator pos;
for (pos=coll.begin(); pos!=coll.end(); ++pos) {
    ...;
}
```

All containers except vectors and deques guarantee that iterators and references to elements remain valid if other elements are deleted. For vectors, only the elements before the point of erase remain valid.

If you remove all elements by using `clear()`, for vectors, deques, and strings any *past-the-end iterator* returned by `end()` or `cend()` may become invalid.

If you insert elements, only lists, forward lists, and associative containers guarantee that iterators and references to elements remain valid. For vectors, this guarantee is given if insertions don't exceed the capacity. For unordered containers, that guarantee is given to references in general but to iterators only when no rehashing happens, which is guaranteed as long as with insertions the number of resulting elements is less than the bucket count times the maximum load factor.

7.1.3. Container Types

All containers provide common type definitions, which are listed in Table 7.3.

Table 7.3. Common Types Defined by All Container Classes

Type	Req	Effect
`size_type`	Yes	Unsigned integral type for size values
`difference_type`	Yes	Signed integral type for difference values
`value_type`	Yes	Type of the elements
`reference`	Yes	Type of element references
`const_reference`	Yes	Type of constant element references
`iterator`	Yes	Type of iterators
`const_iterator`	Yes	Type of iterators to read-only elements
`pointer`	–	Type of pointers to elements (since C++11)
`const_pointer`	–	Type of pointers to read-only elements (since C++11)