8.7. Inserting and Removing Elements

8.7.1. Inserting Single Elements

```
iterator container::insert (const T& value)
iterator container::insert (T&& value)
pair<iterator,bool> container::insert (const T& value)
pair<iterator,bool> container::insert (T&& value)
```

- Insert *value* into an associative or unordered container.
- The first and third forms copy *value*.
- The second and fourth forms move *value* to the container so that the value of *value* is unspecified afterward.
- Containers that allow duplicates, (unordered) multisets and multimaps, have the first and second signatures. They return the position of the new element. Since C++11, newly inserted elements are guaranteed to be placed at the end of existing equivalent values.
- Containers that do not allow duplicates, (unordered) sets and maps, have the third and fourth signatures. If they can't insert the value because an element with an equal value or key exists, they return the position of the existing element and `false` . If they can insert the value, they return the position of the new element and `true` .
- `T` is the type of the container elements. Thus, for (unordered) maps and multimaps, it is a key/value pair.
- For map, multimap, unordered map, and unordered multimap, the corresponding form with move semantic is a member template (see Section 3.2, page 34). Thus, *value* may have any type convertible into the value type (key/value pair) of the container. This was introduced to allow you to pass two strings so that the first one gets converted into a constant string (which is the key type).
- The functions either succeed or have no effect, provided that for unordered containers the hash function does not throw.
- For all containers, references to existing elements remain valid. For associative containers, all iterators to existing elements remain valid. For unordered containers, iterators to existing elements remain valid if no rehashing is forced (if the number of resulting elements is equal to or greater than the bucket count times the maximum load factor).
- The second and fourth forms are available since C++11.
- Provided by set, multiset, map, multimap, unordered set, unordered multiset, unordered map, unordered multimap.

```
iterator container::emplace (args...)
pair<iterator,bool> container::emplace (args...)
```

- Insert a new element initialized by *args* into an associative or unordered container.
- Containers that allow duplicates (ordered and unordered multisets and multimaps) have the first signature. They return the position of the new element. It is guaranteed that newly inserted elements are placed at the end of existing equivalent values.
- Containers that do not allow duplicates (ordered and unordered sets and maps) have the second signature. If they can't insert the value because an element with an equal value or key exists, they return the position of the existing element and `false` . If they can insert the value, they return the position of the new element and `true` .
- The functions either succeed or have no effect, provided that for unordered containers the hash function does not throw.
- For all containers, references to existing elements remain valid. For associative containers, all iterators to existing elements remain valid. For unordered containers, iterators to existing elements remain valid if no rehashing is forced (if the number of resulting elements is equal to or greater than the bucket count times the maximum load factor).
- Note that for sequence containers, the same signature is possible, where the first argument is processed as the position where the new element gets inserted (see Section 8.7.1, page 414).
- Note that to emplace new key/value pairs for (unordered) maps and multimaps, you have to use piecewise construction (see Section 7.8.2, page 341, for details).
- Available since C++11.
- Provided by set, multiset, map, multimap, unordered set, unordered multiset, unordered map, unordered multimap.

Click here to view code image

```
iterator container::insert (const_iterator pos, const T& value)
iterator container::insert (const_iterator pos, T&& value)
```

- Insert *value* at the position of iterator *pos*.
- The first form copies *value*.
- The second form moves *value* to the container so that the value of *value* is unspecified afterward.
- Return the position of the new element.
- If the container does not allow duplicates (set, map, unordered set, unordered map) and already contains an element equal to (the key of) *value*, the call has no effect, and the return value is the position of the existing element.

- For associative and unordered containers, the position is used only as a hint pointing to where the insert should start to search. If *value* is inserted right at *pos*, the function has amortized constant complexity; otherwise, it has logarithmic complexity.
- For vectors, these operations invalidate iterators and references to all elements unless these elements reside before *pos* and no reallocation happens.
- For deques, they invalidate all iterators and, if *pos* is not one of both ends, all references.
- `T` is the type of the container elements. Thus, for (unordered) maps and multimaps, it is a key/value pair.
- For map, multimap, unordered map, and unordered multimap, the second form with move semantics is a member template (see Section 3.2, page 34). Thus, *value* may have any type convertible into the value type (key/value pair) of the container. This was introduced to allow passing two strings so that the first one gets converted into a constant string (which is the key type).
- For strings, *value* is passed by value.
- For vectors and deques, if the copy/move operations (constructor and assignment operator) of the elements don't throw, the functions either succeed or have no effect. For unordered containers, the functions either succeed or have no effect if the hash function does not throw. For all other standard containers, the functions either succeed or have no effect.
- The second form is available since C++11. Before C++11, type `iterator` was used instead of `const_iterator`.
- Provided by vector, deque, list, set, multiset, map, multimap, unordered set, unordered multiset, unordered map, unordered multimap, string.

iterator **container::emplace** (const_iterator *pos*, *args*)

- Inserts a new element initialized by *args* at the position of iterator *pos*.
- Returns the position of the new element.
- For vectors, this operation invalidates iterators and references to all elements unless these elements reside before *pos* and no reallocation happens.
- For deques, it invalidates all iterators and, if *pos* is not one of both ends, all references.
- `T` is the type of the container elements.
- For vectors and deques, if the copy operations (copy constructor and assignment operator) of the elements don't throw, the function either succeeds or has no effect. For all other standard containers, the function either succeeds or has no effect.
- For associative containers, the same signature is possible, where *pos* is processed as first argument for the new element (see Section 8.7.1, page 412).
- The function either succeeds or has no effect, provided that for unordered containers the hash function does not throw.
- Available since C++11.
- Provided by vector, deque, list.

iterator **container::emplace_hint** (const_iterator *pos*, *args*)

- Inserts a new element initialized by *args* at the position of iterator *pos*.
- Returns the position of the new element.
- If the container does not allow duplicates (set, map, unordered set, unordered map) and already contains an element equal to (the key of) *value*, the call has no effect, and the return value is the position of the existing element.
- The position is used only as a hint, pointing to where the insert should start to search. If the new element is inserted at *pos*, the function has amortized constant complexity; otherwise, it has logarithmic complexity.
- `T` is the type of the container elements. Thus, for (unordered) maps and multimaps, it is a key/value pair.
- The function either succeeds or has no effect, provided that for unordered containers the hash function does not throw.
- Available since C++11.
- Provided by set, multiset, map, multimap, unordered set, unordered multiset, unordered map, unordered multimap.

void **container::push_front** (const T& *value*)
void **container::push_front** (T&& *value*)

- Insert *value* as the new first element.
- The first form copies *value*.
- The second form, which is available since C++11, moves *value* to the container, so the state of *value* is undefined afterward.
- `T` is the type of the container elements.
- Both forms are equivalent to `insert(begin(),` *value* `)`.
- For deques, these operations invalidate iterators to all elements. References to elements remain valid.
- These functions either succeed or have no effect (for deques, this guarantee can be given because copy construction is used on reallocation if the move constructor can throw).[1]

[1] The standard currently does not say this, which likely is a defect.

- Provided by deque, list, forward list.

void **container::emplace_front** (*args*)

- Inserts a new first element, which is initialized by the argument list *args*.
- Thus, for the element type there must be a callable constructor for *args*.
- For deques, this operation invalidates iterators to all elements. References to elements remain valid.
- This function either succeeds or has no effect (for deques, this guarantee can be given because copy construction is used on reallocation if the move constructor can throw).[1]
- Available since C++11.
- Provided by deque, list, forward list.

`void` **container::push_back** `(const T&` *value*`)`
`void` **container::push_back** `(T&&` *value*`)`

- Append *value* as the new last element.
- The first form copies *value*.
- The second form, which is available since C++11, moves *value* to the container, so the state of *value* is undefined afterward.
- `T` is the type of the container elements.
- Both forms are equivalent to `insert(end(),` *value* `)`.
- For vectors, these operations invalidate iterators and references to all elements if reallocation happens (the new number of elements exceeds the previous capacity).
- For deques, they invalidate iterators to all elements. References to elements remain valid.
- For strings, *value* is passed by value.
- These functions either succeed or have no effect (for vectors and deques, this guarantee can be given because copy construction is used on reallocation if the move constructor can throw).[1]
- Provided by vector, deque, list, string.

`void` **container::emplace_back** `(`*args*`)`

- Appends a new last element, which is initialized by the argument list *args*.
- Thus, for the element type, there must be a callable constructor for *args*.
- For vectors, this operation invalidates iterators and references to all elements if reallocation happens (the new number of elements exceeds the previous capacity).
- For deques, it invalidates iterators to all elements. References to elements remain valid.
- This function either succeeds or has no effect (for vectors and deques, this guarantee can be given because copy construction is used on reallocation if the move constructor can throw).[1]
- Available since C++11.
- Provided by vector, deque, list.

8.7.2. Inserting Multiple Elements

`void` **container::insert** `(`*initializer-list*`)`

- Inserts copies of the elements of *initializer-list* into an associative container.
- For all containers, references to existing elements remain valid. For associative containers, all iterators to existing elements remain valid. For unordered containers, iterators to existing elements remain valid if no rehashing is forced (if the number of resulting elements is equal to or greater than the bucket count times the maximum load factor).
- Available since C++11.
- Provided by set, multiset, map, multimap, unordered set, unordered multiset, unordered map, unordered multimap.

`iterator` **container::insert** `(const_iterator` *pos*`,` *initializer-list*`)`

- Inserts copies of the elements of *initializer-list* at the position of iterator *pos*.
- Returns the position of the first inserted element or *pos* if *initializer-list* is empty.
- For vectors, this operation invalidates iterators and references to all elements unless these elements reside before *pos* and no reallocation happens.
- For deques, it invalidates all iterators and, if *pos* is not one of both ends, all references.
- For lists, the function either succeeds or has no effect.
- Available since C++11.
- Provided by vector, deque, list, string.

`iterator` **container::insert** `(const_iterator` *pos*`,`
`size_type` *num*`, const T&` *value*`)`

- Inserts *num* copies of *value* at the position of iterator *pos*.
- Returns the position of the first inserted element or *pos* if *num* `==0` (before C++11. nothing was returned).

- For vectors, this operation invalidates iterators and references to all elements unless these elements reside before *pos* and no reallocation happens.
- For deques, it invalidates all iterators and, if *pos* is not one of both ends, all references.
- `T` is the type of the container elements. Thus, for maps and multimaps, it is a key/value pair.
- For strings, *value* is passed by value.
- For vectors and deques, if the copy/move operations (constructor and assignment operator) of the elements don't throw, the function either succeeds or has no effect. For lists, the function either succeeds or has no effect.
- Before C++11, type `iterator` was used instead of `const_iterator` and the return type was `void` .
- Provided by vector, deque, list, string.

```
void container::insert (InputIterator beg, InputIterator end)
```

- Inserts copies of all elements of the range `[` *beg* `,` *end* `)` into the associative container.
- This function is a member template (see Section 3.2, page 34). Thus, the elements of the source range may have any type convertible into the element type of the container.
- For all containers, references to existing elements remain valid. For associative containers, all iterators to existing elements remain valid. For unordered containers, iterators to existing elements remain valid if no rehashing is forced (if the number of resulting elements is equal to or greater than the bucket count times the maximum load factor).
- The function either succeeds or has no effect, provided that for unordered containers the hash function does not throw.
- Provided by set, multiset, map, multimap, unordered set, unordered multiset, unordered map, unordered multimap.

Click here to view code image

```
iterator container::insert (const_iterator pos,
                            InputIterator beg, InputIterator end)
```

- Inserts copies of all elements of the range `[` *beg* `,` *end* `)` at the position of iterator *pos*.
- Returns the position of the first inserted element or *pos* if *beg* `==` *end* (before C++11, nothing was returned).
- This function is a member template (see Section 3.2, page 34). Thus, the elements of the source range may have any type convertible into the element type of the container.
- For vectors, this operation invalidates iterators and references to all elements unless these elements reside before *pos* and no reallocation happens.
- For deques, it invalidates all iterators and, if *pos* is not one of both ends, all references.
- For lists, the function either succeeds or has no effect.
- Before C++11, type `iterator` was used instead of `const_iterator` and the return type was `void` .
- Provided by vector, deque, list, string.

8.7.3. Removing Elements

```
size_type container::erase (const T& value)
```

- Removes all elements equivalent to *value* from an associative or unordered container.
- Returns the number of removed elements.
- Calls the destructors of the removed elements.
- `T` is the type of the sorted value:
  – For (unordered) sets and multisets, it is the type of the elements.
  – For (unordered) maps and multimaps, it is the type of the keys.
- The function does not invalidate iterators and references to other elements.
- The function may throw if the comparison test or hash function object throws.
- Provided by set, multiset, map, multimap, unordered set, unordered multiset, unordered map, unordered multimap.
- For (forward) lists, `remove()` provides the same functionality (see Section 8.8.1, page 420). For other containers, the `remove()` algorithm can be used (see Section 11.7.1, page 575).

```
iterator container::erase (const_iterator pos)
```

- Removes the element at the position of iterator *pos*.
- Returns the position of the following element (or `end()`).
- Calls the destructor of the removed element.
- The caller must ensure that the iterator *pos* is valid. For example:

```
coll.erase(coll.end()); // ERROR ⇒ undefined behavior
```

- For vectors and deques, this operation might invalidate iterators and references to other elements. For all other containers, iterators and references to other elements remain valid.
- For lists, the function does not throw. For vectors and deques, the function does not throw provided the copy/move constructor or assignment operator does not throw. For associative and unordered containers, the function may throw if the comparison test or hash function object throws.
- Before C++11, the return type was `void` for associative containers, and type `iterator` was used instead of `const_iterator`.
- For sets that use iterators as elements, calling `erase()` might be ambiguous since C++11. For this reason, C++11 currently gets fixed to provide overloads for both `erase(iterator)` and `erase(const_iterator)`.
- Provided by vector, deque, list, set, multiset, map, multimap, unordered set, unordered multiset, unordered map, unordered multimap, string.

`iterator` **`container::erase`** `(const_iterator` *beg*, `const_iterator` *end*`)`

- Removes the elements of the range `[` *beg* `,` *end* `)`.
- Returns the position of the element that was behind the last removed element on entry (or `end()`).
- As always for ranges, all elements, including *beg* but excluding *end*, are removed.
- Calls the destructors of the removed elements.
- The caller must ensure that *beg* and *end* define a valid range that is part of the container.
- For vectors and deques, this operation might invalidate iterators and references to other elements. For all other containers, iterators and references to other elements remain valid.
- For lists, the function does not throw. For vectors and deques, the function does not throw provided the copy/move constructor or assignment operator does not throw. For associative and unordered containers, the function may throw if the comparison test or hash function object throws.
- Before C++11, the return type was `void` for associative containers and type `iterator` was used instead of `const_iterator`.
- Provided by vector, deque, list, set, multiset, map, multimap, unordered set, unordered multiset, unordered map, unordered multimap, string.

`void` **`container::pop_front`** `()`

- Removes the first element of the container.
- Is equivalent to

  container`.erase(`*container*`.begin())`

  or for forward lists, to

  container`.erase_after(`*container*`.before_begin())`
- If the container is empty, the behavior is undefined. Thus, the caller must ensure that the container contains at least one element (`!empty()`).
- The function does not throw.
- Iterators and references to other elements remain valid.
- Provided by deque, list, forward list.

`void` **`container::pop_back`** `()`

- Removes the last element of the container.
- Is equivalent to

  container`.erase(prev(`*container*`.end()))`
- If the container is empty, the behavior is undefined. Thus, the caller must ensure that the container contains at least one element (`!empty()`).
- The function does not throw.
- Iterators and references to other elements remain valid.
- For strings, it is provided since C++11.
- Provided by vector, deque, list, string.

`void` **`container::clear`** `()`

- Removes all elements (empties the container).
- Calls the destructors of the removed elements.

- Invalidates all iterators and references to elements of the container.
- For vectors, deques, and strings, it even invalidates any past-the-end-iterator, which was returned by `end()` or `cend()`.
- The function does not throw (before C++11, for vectors and deques, the function could throw if the copy constructor or assignment operator throws).
- Provided by vector, deque, list, forward list, set, multiset, map, multimap, unordered set, unordered multiset, unordered map, unordered multimap, string.

8.7.4. Resizing

```
void container::resize (size_type num)
void container::resize (size_type num, const T& value)
```

- Change the number of elements to *num*.
- If `size()` is *num* on entry, they have no effect.
- If *num* is greater than `size()` on entry, additional elements are created and appended to the end of the container. The first form creates the new elements by calling their default constructor; the second form creates the new elements as copies of *value*.
- If *num* is less than `size()` on entry, elements are removed at the end to get the new size. In this case, they call the destructor of the removed elements.
- For vectors and deques, this operation might invalidate iterators and references to other elements. For all other containers, iterators and references to other elements remain valid.
- For vectors and deques, these functions either succeed or have no effect, provided that the constructor or the assignment operator of the elements doesn't throw. For lists and forward lists, the functions either succeed or have no effect.
- Before C++11, *value* was passed by value.
- For strings, *value* is passed by value.
- Provided by vector, deque, list, forward list, string.