Iterator invalidation rules

What are the iterator invalidation rules for C++ containers?

Preferably in a summary list format.

(Note: This is meant to be an entry to Stack Overflow's C++ FAQ. If you want to critique the idea of providing an FAQ in this form, then the posting on meta that started all this would be the place to do that. Answers to that question are monitored in the C++ chatroom, where the FAQ idea started out in the first place, so your answer is very likely to get read by those who came up with the idea.)

c++ c++11 iterator c++03 c++-faq

edited May 2 '14 at 14:38
Yakk
65.9k 7 51 134

asked Jun 22 '11 at 10:01
Lightness Races in Orbit
177k 30 257 482

4 Answers

C++03 (Source: Iterator Invalidation Rules (C++03))

Insertion

Sequence containers

- `vector` : all iterators and references before the point of insertion are unaffected, unless the new container size is greater than the previous capacity (in which case all iterators and references are invalidated) [23.2.4.3/1]
- `deque` : all iterators and references are invalidated, unless the inserted member is at an end (front or back) of the deque (in which case all iterators are invalidated, but references to elements are unaffected) [23.2.1.3/1]
- `list` : all iterators and references unaffected [23.2.2.3/1]

Associative containers

- `[multi]{set,map}` : all iterators and references unaffected [23.1.2/8]

Container adaptors

- `stack` : inherited from underlying container
- `queue` : inherited from underlying container
- `priority_queue` : inherited from underlying container

Erasure

Sequence containers

- `vector` : every iterator and reference after the point of erase is invalidated [23.2.4.3/3]
- `deque` : all iterators and references are invalidated, unless the erased members are at an end (front or back) of the deque (in which case only iterators and references to the erased members are invalidated) [23.2.1.3/4]
- `list` : only the iterators and references to the erased element is invalidated [23.2.2.3/3]

Associative containers

- `[multi]{set,map}` : only iterators and references to the erased elements are invalidated [23.1.2/8]

Container adaptors

- `stack` : inherited from underlying container
- `queue` : inherited from underlying container
- `priority_queue` : inherited from underlying container

Resizing

- `vector` : as per insert/erase [23.2.4.2/6]
- `deque` : as per insert/erase [23.2.1.2/1]
- `list` : as per insert/erase [23.2.2.2/1]

Note 1

> **Unless otherwise specified** (either explicitly or by defining a function in terms of other functions), invoking a container member function or passing a container as an argument to **a library function shall not invalidate iterators** to, or change the values of, objects within that container. [23.1/11]

Note 2

It's not clear in C++2003 whether "end" iterators are subject to the above rules; you should assume, anyway, that they are (as this is the case in practice).

Note 3

The rules for invalidation of pointers are the sames as the rules for invalidation of references.

edited Nov 1 '11 at 18:55
R. Martinho Fernandes
118k 38 292 394

answered Jun 22 '11 at 10:01
Lightness Races in Orbit
177k 30 257 482

4 Good idea, just to remark: I think that the *associative* containers could be folded together in a single line, and it could be worth then adding another line of the *unordered associative* ones... though I am not sure how the rehashing part could be mapped on insert/erase, do you know of a way to check whether a rehash will be triggered or not ? – Matthieu M. Jun 22 '11 at 10:33

@Matthieu: There's certainly scope here for a bit of C++0x magic. The associative containers were originally semi-folded together, but Martinho changed that and I don't entirely disagree with him. – Lightness Races in Orbit Jun 22 '11 at 10:52

1 IIRC, somewhere the spec says that the end iterator is not an iterator "to objects within that container". I wonder how those guarantees look for the end iterator in each case? – Johannes Schaub - litb Jun 22 '11 at 12:57

1 @MuhammadAnnaqeeb: This answer admittedly doesn't make it clear, as I took a shortcut, but the intention is to say that resizing *is* insertion/erasure, as in if a reallocation is required, you may consider that to be the same as erasing then re-inserting all affected elements. That section of the answer could certainly be improved. – Lightness Races in Orbit Apr 19 at 2:46

1 @Yakk: But it doesn't; see the cited standard text. Looks like that was fixed in C++11 though. :) – Lightness Races in Orbit May 26 at 16:11

C++11 (Source: Iterator Invalidation Rules (C++0x))

Insertion

Sequence containers

- `vector` : all iterators and references before the point of insertion are unaffected, unless the new container size is greater than the previous capacity (in which case all iterators and references are invalidated) [23.3.6.5/1]
- `deque` : all iterators and references are invalidated, unless the inserted member is at an end (front or back) of the deque (in which case all iterators are invalidated, but references to elements are unaffected) [23.3.3.4/1]
- `list` : all iterators and references unaffected [23.3.5.4/1]
- `forward_list` : all iterators and references unaffected *(applies to `insert_after`)* [23.3.4.5/1]
- `array` : *(n/a)*

Associative containers

- `[multi]{set,map}` : all iterators and references unaffected [23.2.4/9]

Unsorted associative containers

- `unordered_[multi]{set,map}` : all iterators invalidated when rehashing occurs, but references unaffected [23.2.5/8]. Rehashing does not occur if the insertion does not cause the container's size to exceed `z * B` where `z` is the maximum load factor and `B` the current number of buckets. [23.2.5/14]

Container adaptors

- `stack` : inherited from underlying container
- `queue` : inherited from underlying container
- `priority_queue` : inherited from underlying container

Erasure

Sequence containers

- `vector` : every iterator and reference at or after the point of erase is invalidated [23.3.6.5/3]
- `deque` : erasing the last element invalidates only iterators and references to the erased elements and the past-the-end iterator; erasing the first element invalidates only iterators and references to the erased elements; erasing any other elements invalidates all iterators and references (including the past-the-end iterator) [23.3.3.4/4]
- `list` : only the iterators and references to the erased element is invalidated [23.3.5.4/3]
- `forward_list` : only the iterators and references to the erased element is invalidated *(applies to `erase_after`)* [23.3.4.5/1]
- `array` : *(n/a)*

Associative containers

- `[multi]{set,map}` : only iterators and references to the erased elements are invalidated [23.2.4/9]

Unordered associative containers

- `unordered_[multi]{set,map}` : only iterators and references to the erased elements are invalidated [23.2.5/13]

Container adaptors

- `stack` : inherited from underlying container
- `queue` : inherited from underlying container
- `priority_queue` : inherited from underlying container

Resizing

- `vector` : as per insert/erase [23.3.6.5/12]
- `deque` : as per insert/erase [23.3.3.3/3]
- `list` : as per insert/erase [23.3.5.3/1]
- `forward_list` : as per insert/erase [23.3.4.5/25]

- `array` : (n/a)

Note 1

> **Unless otherwise specified** (either explicitly or by defining a function in terms of other functions), invoking a container member function or passing a container as an argument to **a library function shall not invalidate iterators** to, or change the values of, objects within that container. [23.2.1/11]

Note 2

> **no swap() function invalidates any references, pointers, or iterators** referring to the elements of the containers being swapped. [Note: **The end() iterator** does not refer to any element, so it **may be invalidated**. —end note] [23.2.1/10]

Note 3

Other than the above caveat regarding `swap()` , it's not clear whether "end" iterators are subject to the above listed per-container rules; you should assume, anyway, that they are.

Note 4

`vector` and all *unordered associative containers* support `reserve(n)` which guarantees that no automatic resizing will occur at least until the size of the container grows to `n` . Caution should be taken with *unordered associative containers* because a future proposal will allow the specification of a minimum load factor, which would allow rehashing to occur on `insert` after enough `erase` operations reduce the container size below the minimum; the guarantee should be considered potentially void after an `erase` .

edited May 26 at 16:13

answered Jun 22 '11 at 15:53

Lightness Races in Orbit
177k 30 257 482

8 +1 found via your blog. On the downvote, dare we guess, people who didn't see the difference (c++0x/c++03) and hand-reflex-voted for duplicate answer? – sehe Nov 14 '11 at 14:32

Beside `swap()` , what's the rules for iterator validity upon copy/move assignment? – goodbyeera Mar 8 '14 at 2:35

@goodbyeera: Copy/move assignment of what? – Lightness Races in Orbit Mar 8 '14 at 11:11

@LightnessRacesinOrbit: Like insertion, erasure, resizing and swap, copy/move assignment are also member functions of std::vector, so I think you could provide the rules of iterator validity for them too. – goodbyeera Mar 8 '14 at 11:17

@goodbyeera: You mean copy/move assigning an element? This will not affect any iterators. Why would it? You're hitting **Note 1** above. – Lightness Races in Orbit Mar 8 '14 at 11:22

It is probably worth adding that an insert iterator of any kind (`std::back_insert_iterator` , `std::front_insert_iterator` , `std::insert_iterator`) is guaranteed to remain valid as long as all insertions are performed through this iterator and no other iterator-invalidating event occurs.

For example, when you are performing a series of insertion operations into a `std::vector` by using `std::insert_iterator` it is quite possible that the vector will experience a reallocation event, which will invalidate all iterators that "point" into that vector. However, the insert iterator in question is guaranteed to remain valid, i.e. you can safely continue the sequence of insertions. There's no need to worry about triggering vector reallocation at all.

This, again, applies only to insertions performed through the insert iterator itself. If iterator-invalidating event is triggered by some independent action on the container, then the insert iterator becomes invalidated as well in accordance with the general rules.

For example, this code

```
std::vector<int> v(10);
std::vector<int>::iterator it = v.begin() + 5;
std::insert_iterator<std::vector<int> > it_ins(v, it);

for (unsigned n = 20; n > 0; --n)
  *it_ins++ = rand();
```

is guaranteed to perform a valid sequence of insertions into the vector, even if the vector "decides" to reallocate somewhere in the middle of this process.

edited Sep 23 '13 at 18:54

answered Jul 4 '12 at 23:36

AnT
177k 19 250 474

Since this question draws so many votes and kind of becomes an FAQ, I guess it would be better to write a separate answer to mention one significant difference between C++03 and C++11 regarding the impact of `std::vector` 's insertion operation on the validity of iterators and references with respect to `reserve()` and `capacity()` , which the most upvoted answer failed to notice.

C++ 03:

> Reallocation invalidates all the references, pointers, and iterators referring to the elements in the sequence. It is guaranteed that no reallocation takes place during insertions that happen after a call to reserve() until the time when an insertion would make the size of the vector **greater than the size specified in the most recent call to reserve()**.

C++11:

> Reallocation invalidates all the references, pointers, and iterators referring to the elements in the sequence. It is guaranteed that no reallocation takes place during insertions that happen after a call to reserve() until the time when an insertion would make the size of the vector **greater than the value of capacity()**.

So in C++03, it is not " `unless the new container size is greater than the previous capacity (in which case all iterators and references are invalidated)` " as mentioned in the other answer, instead, it should be " `greater than the size specified in the most recent call to reserve()` ". This is one thing that C++03 differs from C++11. In C++03, once an `insert()` causes the size of the vector to reach the value specified in the previous `reserve()` call (which could well be smaller than the current `capacity()` since a `reserve()` could result a bigger `capacity()` than asked for), any subsequent `insert()` could cause reallocation and invalidate all the iterators and references. In C++11, this won't happen and you can always trust `capacity()` to know with certainty that the next reallocation won't take place before the size overpasses `capacity()` .

In conclusion, if you are working with a C++03 vector and you want to make sure a reallocation won't happen when you perform insertion, it's the value of the argument you previously passed to `reserve()` that you should check the size against, not the return value of a call to `capacity()` , otherwise you may get yourself surprised at a "*premature*" reallocation.

edited Mar 8 '14 at 11:59

answered Mar 8 '14 at 2:24

neverhoodboy
684 3 7

2 However, I'd shoot any compiler who did this to me, and no jury in the land would convict me. – Yakk May 2 '14 at 14:39

I didn't "fail to notice" this; it was an editorial error in C++03 that was corrected in C++11. No mainstream compiler takes advantage of the error. – Lightness Races in Orbit Nov 13 '14 at 23:49