# What is iterator invalidation?



I see it referenced a lot but no clear answer of what exactly it is. My experience is with higher level languages, so I'm unfamiliar about the presence of invalidity in a collections framework.

What is iterator invalidation?

Why does it come up? Why is it difficult to deal with?

c++ iterator

edited Jun 3 '13 at 21:32
user529758

asked Jun 3 '13 at 19:32

Mark Canlas
**5,145** 20 51

2 Also this Iterator Invalidation Rules – mwerschy Jun 3 '13 at 19:36

I might need a little better explanation of this, I didn't think it had anything to do with high/low level languages. I know you can't modify the list during iteration in `C#` . – Nick Freeman Jun 3 '13 at 19:36

4 Voting to reopen... "what is" is a different question that isn't at all answered in linked thread. – djechlin Jun 3 '13 at 19:40

@NickFreeman it has nothing to do with high vs low levels. It has everything to do with implementation. It is entirely possible (although complex) to create a container in C# which would allow for iteration while enumerating, so long as you keep the state of each in-check. – Richard J. Ross III Jun 3 '13 at 19:40

@RichardJ.RossIII Thanks, that's what I thought. The OP confused me by making it seem like it was only a low level language concept. – Nick Freeman Jun 3 '13 at 19:43

## 3 Answers

1. Iterators are glorified pointers. Iterator invalidation is a lot like pointer invalidation; it means it suddenly points to junk data.

2. Because it's very natural but wrong to do things like this:

```
for(iterator it = map.begin(); it != map.end(); ++it) {
    map.erase(it->first);
    // whoops, now map has been restructured and iterator
    // still thinks itself is healthy
}
```

3. Because that error right there? No compiler error, no warning, you lose. You just have to be trained well enough to watch for them and prevent them. Very insidious bugs if you don't know what you're doing. One of the design philosophies of C++ is speed over safety. The runtime check that would lead iterator invalidation to an exception instead of unspecified behavior is too expensive, in the view of C++ language designers.

You should be on high alert when you are iterating over a data structure and *modifying* structure itself, instead of merely the objects held in them. At that point you should probably run to the documentation and check whether the operation was illegal (For instance, someone in the comments did exactly that, and discovered the iterator above is still healthy. Oops to me. Good work, citizen.)

edited Feb 7 at 19:39

answered Jun 3 '13 at 19:36

djechlin
**23.1k** 11 52 121

6 This is why I love the fact that Visual Studio has debug builds. By default, the standard C++ containers will check if they're valid in debug builds, helping to find these insidious errors. – Mooing Duck Jun 3 '13 at 19:41

@MooingDuck you shouldn't rely upon that behavior, though. If you become too attached to awesome compiler features, when you move over to a different platform (which may happen some time in the future), you will get burned, hard and fast. – Richard J. Ross III Jun 3 '13 at 19:42

3 @RichardJ.RossIII: Depend on? Clearly not. But it's nice to have an extra layer of protection for when I do screw up. – Mooing Duck Jun 3 '13 at 19:52

But what kind of iterator implementation lets that happen at all? Isn't that kind of ghetto/unsafe? – Mark Canlas Jun 3 '13 at 19:59

6 @MarkCanlas: That's how C++ works. Most errors that can only be caught at runtime aren't required to be checked automatically; otherwise, the runtime cost would be unacceptable in many situations. So a "safe" implementation (as Mooing Duck describes) will check for this error; a "fast" implementation won't, and will be much faster. – Mike Seymour Jun 3 '13 at 20:17



Iterator invalidation is what happens when an iterator type (an object supporting the operators `++` , and `*` ) *does not correctly represent the state of the object it is iterating*. For example:

```
int *my_array = new int[15];
int *my_iterator = &my_array[2];

delete[] my_array;

std::for_each(my_iterator, my_iterator + 5, ...); // invalid
```

That results in undefined behavior because the memory it is pointing to has been reclaimed by the OS.

This is only one scenario, however, and many other things cause an iterator to be 'invalidated', and you must be careful to check the documentation of the objects you are using.

answered Jun 3 '13 at 19:36

Richard J. Ross III

**37.6k** 14 83 137

---

The problem occurs when a container that is being processed using an iterator has its shape changed during the process. (We will assume a single-threaded application; concurrent access to a mutable container is a whole 'nother can of worms which we won't get into on this page). By "having its shape change", one of the following types of mutations is meant:

- An insertion into the container (at any location)
- Deletion of an element from the container

- Any operation that changes a key (in an AssociativeContainer)
- Any operation which changes the order of the elements in a sorted container.
- Any more complicated operation consisting of one or more of the above (such as splitting a container into two).

(From: http://c2.com/cgi/wiki?IteratorInvalidationProblem)

The concept is actually fairly simple, but the side-effects can be quite annoying. I would add that this problem affects not only C/C++ but slews of other low-level or mid-level languages, as well. (In some cases, even if they don't allow direct heap allocation)

edited Jun 3 '13 at 20:59

djechlin
**23.1k** 11 52 121

answered Jun 3 '13 at 19:37

David Titarenco
**19.6k** 8 38 85

Keys can't be changed in any associative container that I know of. – Mooing Duck Jun 3 '13 at 19:42

@MooingDuck it really depends on what you consider a 'key change' to be. I could remove the value for one key, and add that value for another key, isn't that a 'key change'? Just because containers don't natively support something, doesn't mean it can't be done. – Richard J. Ross III Jun 3 '13 at 19:44

@RichardJ.RossIII: I believe the post/link are referring to actual modification of the *key itself*, not the value associated with said key. Modifying the associated data require iterator invalidation in most cases. Modifying the *key itself* would require a (theoretical) erasure/reinsertion. – Mooing Duck Jun 3 '13 at 19:46

@MooingDuck once again, not necessarily. If my implementation is an array of keys, and an array of values, where the index of a value corresponds to it's key, I could simply change the value in the key array! Similarly, if it's an array of key-value pairs, nothing says I can't just change the 'key' portion of the pair. It's obviously not implemented in the C++ STL, but it's not impossible. – Richard J. Ross III Jun 3 '13 at 19:48

@RichardJ.RossIII: I see your point. I mean, that would scale poorly, but that's irrelevant to your point. I can even see how that could invalidate an iterator! O.o – Mooing Duck Jun 3 '13 at 19:54