is insert() necessary in a map or unordered_map?

I see a lot of examples that add items to a `map` or `unordered_map` via `operator[]`, like so:

```
int main() {
    unordered_map <string, int> m;
    m["foo"] = 42;
    cout << m["foo"] << endl;
}
```

Is there any reason to use the `insert` member function instead? It would appear they both do the same thing.

`c++`

edited Mar 5 '14 at 22:17 Kiril Kirov 21.5k 7 66 127

asked Apr 3 '13 at 14:00 hellofunk 8,483 5 46 125

4 Answers

They are not.

`operator[]` will overwrite the value for this key, if it exists, while `insert` will not.

In case `operator[]` is used for inserting element, it is expected to be a little slower (see @MatthieuM's comment below for details), but this is not that significant here.

While `std::map::insert` returns `std::pair< iterator, bool >`, where the `.second` will tell you if the value is inserted or it already exists.

Regarding your comment: you cannot have 2 elements with the same key and different value. This is not a `multimap`.

If there's an element in the map, with the same key you're trying to insert, then:

- `operator[]` will overwrite the existing value
- `std::map::insert` will ~~not do anything.~~* return a `std::pair< iterator, bool >`, where the `.second` will be `false` (saying "the new element is not inserted, as such key already exists") and the `.first` will point to the found element.

* I changed this thanks to the note/remark, given from @luk32; but by writing "will not do anything", I didn't mean it literally, I meant that it will not change the value of the existing element

edited Apr 4 '13 at 7:29

answered Apr 3 '13 at 14:02 Kiril Kirov 21.5k 7 66 127

1 Ok, so what happens if you insert two items with the same key? – hellofunk Apr 3 '13 at 14:02

@SebbyJohanns - see my edit, regarding your comment. – Kiril Kirov Apr 3 '13 at 14:05

1 The data for that key will not be overwritten. The bracket notation is equivalent to `(* ((m.insert(value_type(k, data_type()))).first)).second = data;` (borrowed from footnote 3 here). Which is to say, it inserts the key with a default-constructed datum, then takes the iterator returned (the `.first`), which either points to the old, unmodified key-value pair, if it existed, or to the new one just added, and sets the data (the `.second`) associated with the key to your desired value. – metal Apr 3 '13 at 14:07

`operator[]` will not do anything either. `T::operator=` Using the reference from obtained from `map::operator[]` will do. And you can obtain the same reference using `insert()` . It is down to strict semantic analysis. But I think I am right here, while I still understand your more human-like approach, where it does seem that they behave differently. But when you break down the code, they are not. It's just more stuff happens here `mymap["answer"]=42` than it seems. At least to my feeling. – luk32 Apr 3 '13 at 14:54

1 @KirilKirov: `operator[]` is expected to be a little slower, but I would not say it is because of the check; both `operator[]` and `insert` perform the same number of checks. `operator[]` is slower for straight insertion because it first builds a default value *and then* use assignment to overwrite it. – Matthieu M. Apr 4 '13 at 7:22

Using `insert()` can help improve performance in certain situations (more specifically for `std::map` since search time is `O(log(n))` instead of constant amortized). Take the following common example:

```
std::map<int, int> stuff;

// stuff is populated, possibly large:

auto iterator = stuff.find(27);

if(stuff.end() != iterator)
{
   // subsequent "find", set to 15
   iterator->second = 15;
}
else
{
   // insert with value of 10
   stuff[27] = 10;
}
```

The code above resulted in effectively finding the element twice. We can make that (slightly) more efficient written like this:

```
// try to insert 27 -> 10
auto result = stuff.insert(std::make_pair(27, 10));

// already existed
if(false == result.second)
{
   // update to 15, already exists
   result.first->second = 15;
}
```

The code above only tries to find an element once, reducing algorithmic complexity. For frequent operations, this can improve performance drastically.

answered Apr 3 '13 at 14:28

Chad
11.1k 1 18 33

Well I disagree with Kiril's answer to a certain degree and I think it's not full so I give mine.

According to `cppreference` `std::map::operator[]` is equivalent to a certain `insert()` call. By this I also think he is wrong saying the value will be overwritten. It says: "Return value Reference to the mapped value of the new element if no element with key key existed. Otherwise a reference to the mapped value of the existing element is returned."

So it seems it is a convenient wrapper. The `insert()` , however has this advantage of being overloaded, so it provides more functionality under one name.

I give a point to Kiril, that they do seem to have a bit different functionality at first glance, however IHMO the examples he provides are not equivalent to each other.

Therefore, as an example/reason to use `insert` I would point out, inserting many elements at once, or using `hint` (Calls 3-6 in here).

So is insert() necessary in a map or unordered_map? I would say yes. Moreover, the

`operator[]` is not necessary as it can be emulated/implemented using `insert`, while the other way is impossible! It simply provides more functinality. However, writing stuff like `(insert(std::make_pair(key, T())).first)->second)` (after cppreference) seems cumbersome than `[]`.

Thus, is there any reason to use the insert member function instead? I'd say for overlapping functionality, hell no.

edited Apr 3 '13 at 14:48

answered Apr 3 '13 at 14:16

luk32

7,654 8 29

I agree, but returning a reference and using this code `m["foo"] = 42` (from the question) actually overwrites the value of `foo` if it existed before this call, right? Also, I don't say, that one of these `insert` or `operator[]` are better. Each of them has their own advantages (I'm saying this because of your note for the overloaded range `insert`). What I mean is - they can be used differently, but the OP asked for a specific case. +1 anyway, good points. – Kiril Kirov Apr 3 '13 at 14:21

OK, it will effectively overwrite it. That's why I said I disagree to a certain degree. Still I am not sure if you can say it is entirely different, as you can implement `operator[]` with `insert()`. I'd say your 2nd part of answer is murky to say the least. `m["foo"]` will do nothing. Just as `*(insert("foo")->first) = 42` will update value, and this is the equivalent to your example. I think you confused two things. But I understand your point as well. – luk32 Apr 3 '13 at 14:33

I never said, that `operator[]` cannot be implemented using `insert`. If you're talking about my words "will not do anything" - OK, I agree that it's not exactly "nothing", but I didn't mean it *literally*. OK, I'll edit it and I'll give you a reference, as it could be misunderstood badly. – Kiril Kirov Apr 3 '13 at 14:39

The two are not equivalent. `insert` will not overwrite an existing value, and it returns a `pair<iterator, bool>`, where `iterator` is the location of the key, regardless of whether or not it already existed. The `bool` indicates whether or not the insert occurred.

`operator[]` effectively does a `lower_bound` on key. If the result of that operation is an `iterator` with the same key, it returns a reference to the value. If not, it inserts a new node with a default-constructed value, and then returns a reference to the value. This is why `operator[]` is a non-const member - it auto-vivifies the key-value if it doesn't exist. **This may have performance implications if the value type is costly to construct.**

Also note in C++11, we have an `emplace` method that works nearly identical to `insert`, except it constructs the key-value pair in-place from forwarded arguments, if an insert occurs.

answered Apr 3 '13 at 15:40

Nathan Ernst

3,212 9 28