# A simplified overview of ways to add or update elements in a std::map

devblogs.microsoft.com/oldnewthing/20250113-00

January 13, 2025


Some time ago, I mentioned how the `std::map` subscript operator is a dangerous convenience. In that article, I linked to an overview of the insertion emplacement methods, but I'm going to recapture the essential points in a table.[1]

In the table below, the discussion of "consumed" or "not consumed" refers to the case that `v` is an rvalue reference like `std::move(something)`.

| Statement | If key present | If key not present |
|---|---|---|
| `m.insert({ k, v });` | No effect, `v` is consumed | Added, `v` is consumed |
| `m.emplace(k, v);` | No effect, `v` is consumed | Added, `v` is consumed |
| `m.try_emplace(k, v);` | No effect, `v` is not consumed | Added, `v` is consumed |
| `m.insert_or_assign(k, v);` | Updated, `v` is consumed | Added, `v` is consumed |
| `m.at(k) = v;` | Updated, `v` is consumed | Throws, `v` is not consumed |

We can reorganize the table by effect.

| | | If key present | | |
|---|---|---|---|---|
| | | **No effect<br>`v` not consumed** | **No effect<br>`v` consumed** | **Updated<br>`v` consumed** |
| **If key not present** | **Added** | `m.try_emplace(k, v);` | `m.insert({ k, v });`<br>`m.emplace(k, v);` | `m.insert_or_assign(k, v);` |
| | **Throws** | | | `m.at(k) = v;` |

**Exercise**: Why are the bottom left two boxes blank?

**Sidebar**: I intentionally omit `m[k] = v;` as a possibility because behaves the same as `insert_or_assign`, but with worse performance, and works in fewer circumstances: If the key does not exist, then `m[k] = v` first creates a default-constructed value and inserts it into the map, and then overwrites that default-constructed value with `v`. This creates an empty value unnecessarily, and it requires that the value type be default-constructible. **End sidebar**

Often overlooked is that the "no effect if key is present" methods also tell you whether it did anything. This means that you can avoid double-probing the map.

```
// inefficient version
if (map.contains(k)) {
    already_present();
} else {
    map[k] = v;
    newly_added();
}

// more efficient alternative
// (assuming v is easy to calculate)
if (map.try_emplace(k, v).second) {
    newly_added();
} else {
    already_present();
}
```

Instead of checking whether the map contains a key before adding the element, just try to add it and deal with the collision.

Here's another simplification:

```
// inefficient version
do {
    k = gen_random_key();
} while (map.contains(k));
map[k] = v;

// more efficient alternative
do {
    k = gen_random_key();
while (!map.try_emplace(k, v).second);
```

Again, instead of checking whether the map contains a key before adding the element, just try to add it and deal with the collision.

For completeness, here's a table for reading from a map.

| Statement | If key present | If key not present |
|---|---|---|
| `auto& v = m[k];` | Reference to existing value | Reference to newly-created value |
| `auto& v = m.at(k);` | Reference to existing value | Throws |

Another mistake I see is code where each line makes sense on its own, but they perform duplicate work that could be combined.

```
// inefficient version
if (m.find(k) != m.end()) {
    m[k].SomeMethod();
}

// more efficient alternative
if (auto found = m.find(k); found != m.end()) {
    found->second.SomeMethod();
}
```

The two pieces make sense separately. The `m.find(k) != m.end()` is the standard way to detect whether a key is present in the map. And the `m[k].SomeMethod()` is a standard way to invoke a method on a value stored in the map (though really, should be `m.at(k).SomeMethod()`). But the second statement repeats work done by the first statement, because `m.at(k)` and `m[k]` are just a `m.find(k)->second`, with an exception if the item is not found (for `m.at()`) or creating the entry if the item is not found (for `m[]`). If you've already done the `m.find(k)` in the `if` statement, you can use that result instead of making `m.at()` and `m[]` search for it a second time. (On top of that, `m[]` has to create an empty entry if the key is missing.)

I've also seen code that doesn't realize that the `[]` operator creates an empty entry if the key is not present. This manifests in two ways:

```
std::map<Key, std::vector<Value>> m;

// problem 1: Creating unnecessary empty entries
m[k] = new_value;

// problem 2: Manually creating empty entries
std::vector<Value> strings;
if (auto found = m.find(k); found != m.end())
{
    strings = found->second;
}
strings.push_back(new_value);
m[k] = strings;

// better version of 1: Use insert_or_assign
m.insert_or_assign(k, new_value);

// better version of 2: Create the empty entry on demand
m[k].push_back(new_value);
```

**Bonus chatter**: In retrospect, there probably shouldn't have been a `[]` operator for `std::map`, since it takes the most convenient syntax for a rarely-needed operation. It should have been called something like `m.create_or_get(k)`.

**Answer to exercise**: The bottom left two boxes correspond to hypothetical operations that throw if the key is missing, and do nothing if the key is present. This is not very useful, so the standard provides no method to perform it. I guess you could use `(void)m.at(k)`.

[1] More accurately, in another table.