The operations for reading and writing single elements for C++ standard library maps

devblogs.microsoft.com/oldnewthing/20241118-00

November 18, 2024

Some time ago, I noted that the `std::map` subscript operator is an attractive nuisance. It is the most convenient syntax, but is not often what you actually want.

I've broken down the various `std::map` lookup and update operations into a table so you can choose the best one for your situation.

Operation	Method
Read, throw if missing	`m.at(key)`
Read, allow missing	`m.find(key)`
Read, create if missing	`m[key]`

Write, nop if exists, discard value	`m.insert({ key, value })`
	`m.emplace(key, value)`

Write, nop if exists	`m.emplace(std::piecewise_construct, ...)`
	`m.try_emplace(key, params)`

Write, overwrite if exists	`m.insert_or_assign(key, value)`

In the table above, `key` is the map key, `value` is the mapped type, and `params` are parameters to the mapped type constructor.

Note that `insert` and the first `emplace`[1] take a value which is discarded if it turns out that the key already exists. This is undesirable if creating the value is expensive.

One frustrating scenario is the case where the mapped type's default constructor is not the constructor you want to use for `operator[]`, or if you want the initial mapped value to be the result of a function call rather than a constructor. Here's something I sort of threw together.

```
template<typename Map, typename Key, typename Maker>
auto& ensure(Map&& map, Key&& key, Maker&& maker)
{
    auto lower = map.lower_bound(key);
    if (lower == map.end() || !map.key_comp()(lower->first, key)) {
        lower = map.emplace_hint(lower, std::forward<Key>(key),
                                 std::forward<Maker>(maker)());
    }
    return lower->second;
}
```

This returns a reference to the mapped value that corresponds to the provided key, creating one if necessary via the `maker`. We forward the key into `emplace_hint` so it can be moved.

```
struct Widget
{
    Widget(std::string name);

    ⟦ ... other stuff ... ⟧
};
std::map<std::string, std::shared_ptr<Widget>> widgets;

auto& ensure_named_widget(std::string const& name)
{
    return ensure(widgets, name,
        [&] { return std::make_shared<Widget>(name); });
}
```

As a convenience, we can accept bonus parameters which are passed to the `maker`. This allows you to parameterize the `maker`.

```
template<typename Map, typename Key, typename... Maker>
auto& ensure(Map&& map, Key&& key, Maker&&... maker)
{
    auto lower = map.lower_bound(key);
    if (lower == map.end() || map.key_comp()(lower->first, key)) {
        lower = map.emplace_hint(lower, std::forward<Key>(key),
            std::invoke(std::forward<Maker>(maker)...));
    }
    return lower->second;
}

auto& ensure_named_widget(std::string const& name)
{
    return ensure(widgets, name,
        [](auto&& name) { return std::make_shared<Widget>(name); },
        name);
}

// or

auto shared_widget_maker(std::string const& name)
{
    return std::make_shared<Widget>(name);
}

auto& ensure_named_widget(std::string const& name)
{
    return ensure(widgets, name, shared_widget_maker, name);
}
```

But wait, we learned last time that conversion operators are in play. We can use our `EmplaceHelper` in conjunction with `try_emplace`.

```
template<typename Map, typename Key, typename... Maker>
auto& ensure(Map&& map, Key&& key, Maker&&... maker)
{
    return *map.try_emplace(key, EmplaceHelper([&] {
        return std::invoke(std::forward<Maker>(maker)...);
    }).first;
}
```

The `try_emplace` function returns a `std::pair` of an iterator to the matching item (either pre-existing or freshly-created) and a `bool` that indicates whether the item is freshly-created. We don't care about whether the item is freshly-created, so we just take the iterator (the `.first`) and dereference it to get a reference to the item.

This is simple enough that you might just write it out rather than calling out to a one-liner helper.

```
auto& item =
    *widgets.try_emplace(name, EmplaceHelper([&] {
        return std::make_shared<Widget>(name); }).first;

// or

auto& item =
    *widgets.try_emplace(name,
        EmplaceHelper(shared_widget_maker, name)).first;
```

[1] Technically, you can pass the first `emplace` a second parameter that is used to *construct* the `value`, so if your mapped type has a single-parameter constructor, you can use the first `emplace` to get the same effect as the piecewise constructor: on-demand construction of the mapped type.